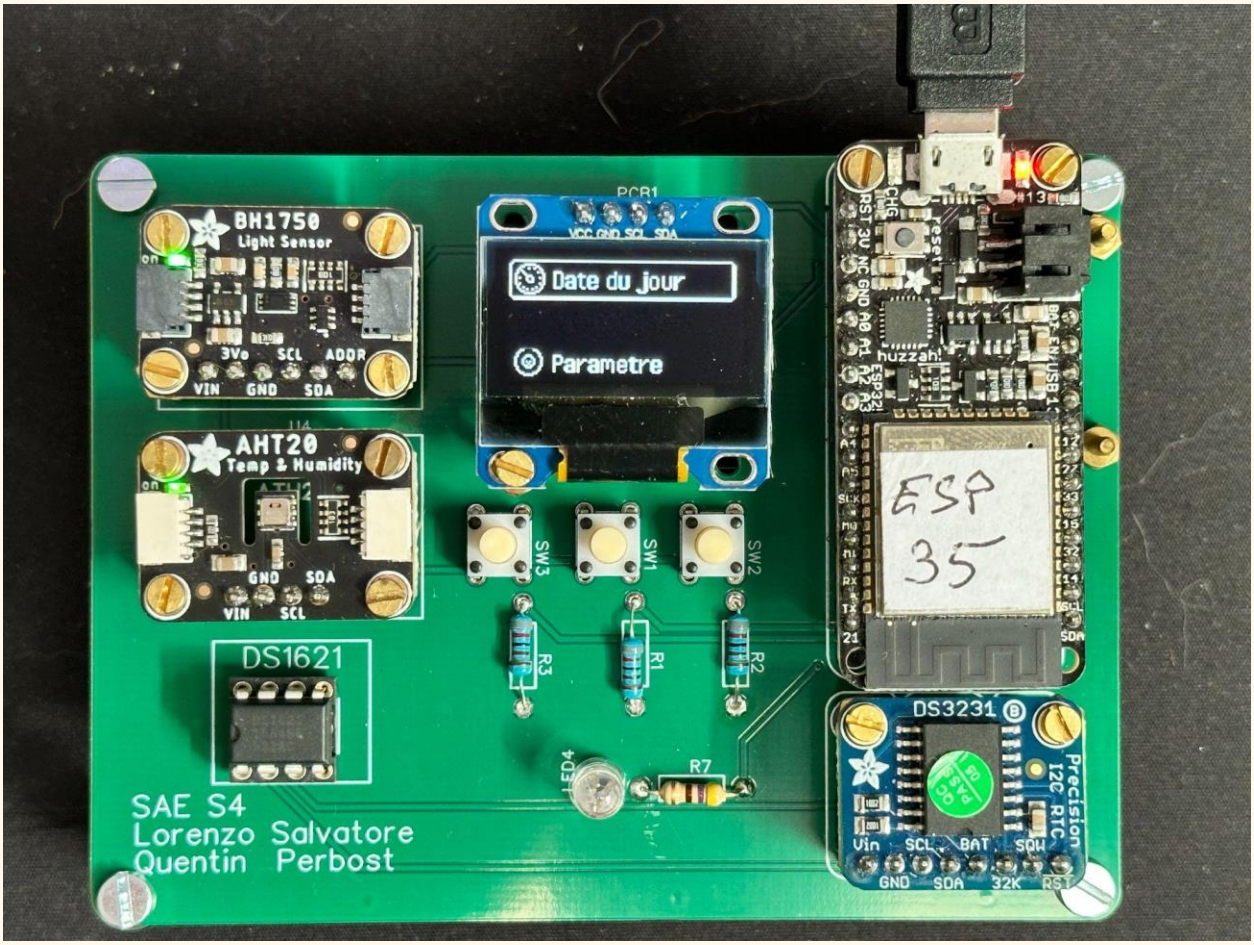
**SAE** 4ème SEMESTRE
MR SALVATORE & MR PERBOST

# Station Météorologique Connectée avec Interface Web

## BUT GEII S4 FA ESE

___ **Département Génie Électrique et Informatique Industrielle - Site de Neuville**



# Sommaire

# I- Introduction générale

Dans le cadre du BUT GEII (Génie Électrique et Informatique Industrielle), nous avons réalisé un projet pratique appelé SAÉ (Situation d'Apprentissage et d'Évaluation), dont le but était de concevoir une station météo connectée. L'objectif principal était de créer un système capable de mesurer différentes données météorologiques, comme la température, l'humidité de l'air, la luminosité ambiante, la date et l'heure, puis d'afficher ces informations à la fois sur un écran OLED et sur une page web accessible via WiFi.

Pour réaliser ce projet, nous avons utilisé une carte ESP32, qui est un microcontrôleur avec WiFi intégré. Nous avons connecté différents capteurs (température, humidité, luminosité, horloge RTC) pour collecter les données. Ensuite, nous avons programmé la carte avec le langage C/C++ en utilisant la plateforme Visual Studio Code, avec l'extension PlatformIO. Une interface web a aussi été créée pour permettre de consulter les données à distance depuis un navigateur.

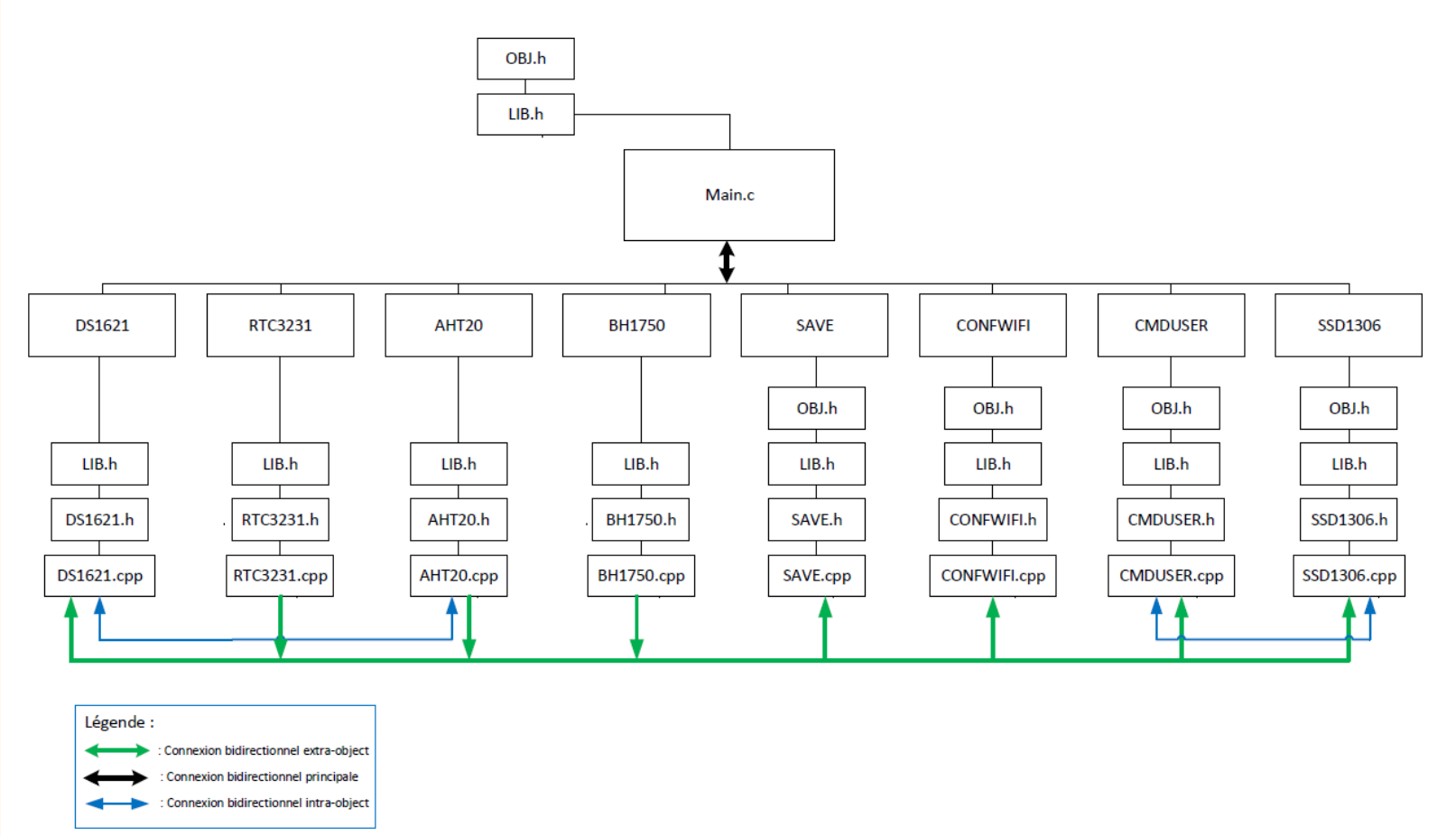
Ce projet nous a permis de mettre en pratique nos connaissances en électronique, en programmation, en communication entre composants (bus $I^2C$), mais aussi d'apprendre à gérer un projet sur plusieurs semaines : planification, répartition des tâches, tests, résolution de bugs, etc. Le travail en équipe a été essentiel, car chacun a pu contribuer sur une partie du projet (capteurs, affichage, interface web...).

Dans ce rapport, nous allons décrire les différentes étapes du projet, les composants utilisés, le fonctionnement général de la station météo, les choix techniques réalisés, ainsi que les difficultés rencontrées et les solutions apportées.

## I-1. Architecture du code

L'architecture du code a été pensée de manière à séparer les différentes fonctions du projet : la lecture des capteurs, l'affichage sur l'écran OLED, la gestion de l'heure, la connexion au WiFi et la création de l'interface web. Chaque partie est organisée en blocs clairs pour faciliter la compréhension et la maintenance du programme.

Le diagramme d'interconnexion ci-dessous vous permettra de bien situer comment hiérarchiser nos différentes bibliothèques et comment elles sont implémentées dans le code.

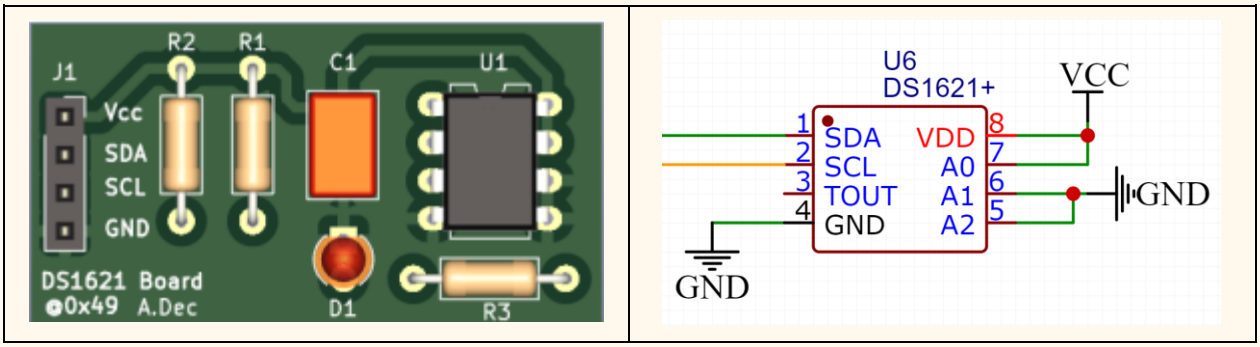


# II- Capteur de température

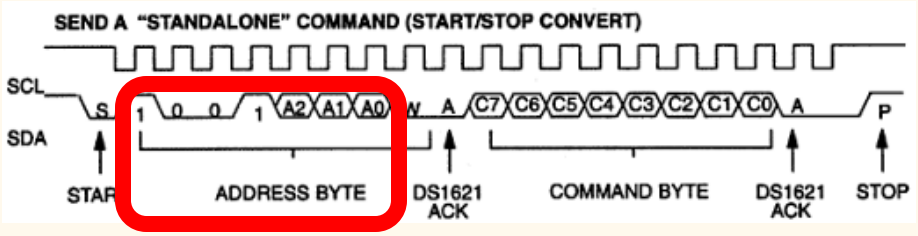
## II-1. Prise en main du capteur DS1621

Pour le capteur de température, nous allons utiliser le capteur DS1621. Le rôle de ce capteur est de mesurer la température ambiante.

Le capteur prendra la forme d'une plaquette comme ci-dessous :




L'adresse I²C de ce capteur est configurée de la manière suivante :

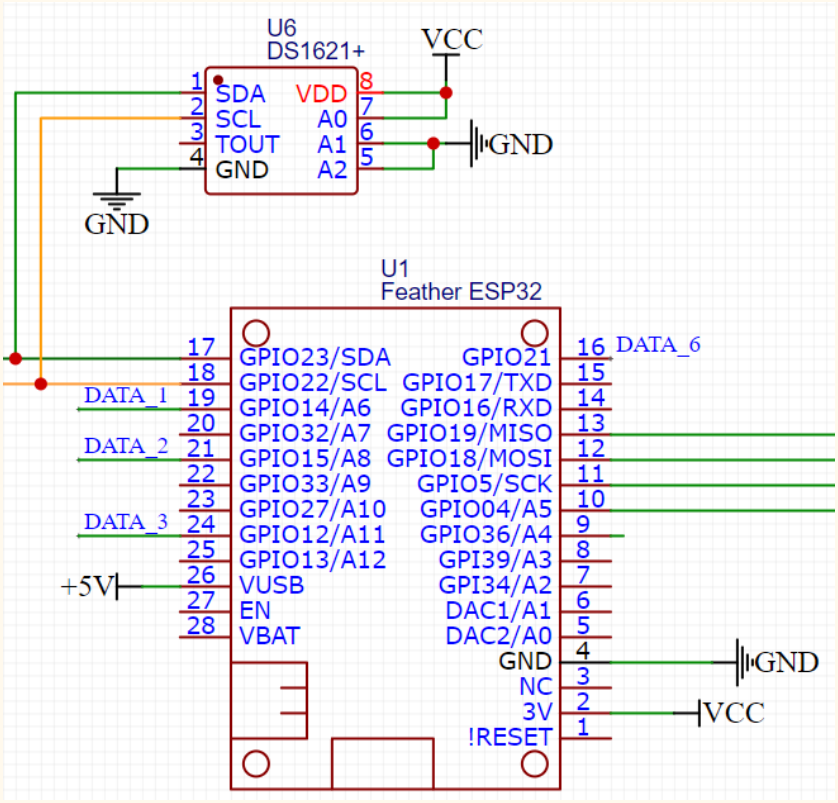


| Data | A6 | A5 | A4 | A3 | A2 | A1 | A0 | Adresse |
|---|---|---|---|---|---|---|---|---|
| Binaire | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0b1001001 |
| Hexadécimal | 4 | | | 8 | | | 1 | 0x49 |

Ce capteur, d'une plage de mesure de température comprise entre -55°C et +125°C, est muni de différents registres comme décrit dans le tableau ci-dessous.

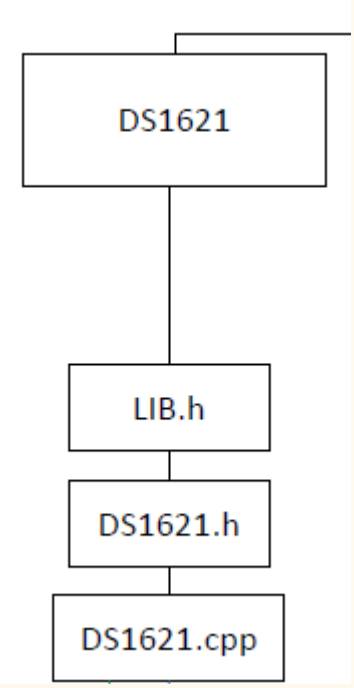
| Registre | Description |
|---|---|
| 0xAC | Accès au registre de configuration (lecture ou écriture) |
| 0xEE | Début de conversion |
| 0x22 | Arrêt conversion. |
| 0xAA | lecture de la température. DS1621 renvoi 2 octets. |
| 0xA1 | Lecture ou écriture du seuil haut du thermostat : TH. |
| 0xA2 | Lecture ou écriture du seuil bas du thermostat : TL |

Le capteur sera relié à l'ESP32 de la manière suivante :




## II-2. Création de la bibliothèque DS1621

La bibliothèque DS1621 est organisée de la manière suivante :




LIB.h permet d'inclure les bibliothèques suivantes :

```
#ifndef LIB_H
#define LIB_H

#include <Ticker.h>
#include <WiFi.h>
#include <LittleFS.h>
#include <ESPAsyncWebServer.h>
#include <Wire.h>
#include <Arduino.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#endif
```

## A. DS1621.h

DS1621.h permet d'initialiser la classe DS1621 et d'y intégrer les différentes méthodes du programme.

```
#ifndef DS1621_H
#define DS1621_H
#include <LIB.h>
// les registres
#define DS1621_start_reg 0xEE
#define DS1621_Read_Temperature 0xAA
#define DS1621_stop_reg 0x22
#define DS1621_config 0xAC
class DS1621{
public:
    // constructeur
    DS1621();
    // méthodes
    bool begin(uint8_t DS1621_ADR);//Initialisation du capteur
    float getValue();//Prise de la température ambiante
    void startConv();//On commence la conversion de la température
    void stopConv();//On stoppe la conversion de température
private:
    uint8_t _address;
    float _temperature;};
#endif
```

### B. DS1621.cpp

Il représente la partie logicielle du projet dédiée à la communication avec le capteur de température DS1621 en langage **C++**.

Ce module a pour rôle principal :

- **D'interroger le capteur** afin de récupérer les données de température mesurées.
- **De configurer les registres internes** du capteur selon les besoins du système (ex. : seuils d'alarme, mode de fonctionnement, démarrage de la conversion, etc.).
- **De fournir une interface claire** entre le capteur matériel et le reste du programme, facilitant son intégration et sa maintenance.

L'implémentation inclut les fonctions nécessaires à la lecture/écriture sur le bus $I^2C$, le traitement des données brutes reçues, ainsi que la gestion des registres spécifiques du DS1621.

La fonction begin :

```
bool DS1621::begin(uint8_t address)
{
    _address = address; // chaque objet aura sa propre adresse
    Wire.begin();
    Wire.beginTransmission(_address);
    if (Wire.endTransmission() == 0)
    {
        Serial.println("Capteur DS1621 connecté.");
        return true;
    }
    else
    {
        Serial.println("Échec de connexion au capteur DS1621 !");
        return false;
    }
}
```

La fonction DS1621::begin(uint8_t address) initialise la communication $I^2C$ avec le capteur DS1621 en utilisant l'adresse fournie en paramètre. Elle commence par stocker cette adresse dans l'attribut _address pour permettre à chaque objet DS1621 d'avoir sa propre configuration. Ensuite, elle initialise le bus $I^2C$ avec Wire.begin(), puis tente d'ouvrir une communication avec le capteur via Wire.beginTransmission(_address). Si Wire.endTransmission() retourne 0, cela signifie que le capteur est bien connecté : un message de confirmation est affiché sur le moniteur série et la fonction retourne true. Dans le cas contraire, un message d'erreur est affiché et la fonction retourne false.

La fonction getValue :

```
float DS1621::getValue(){
    uint8_t tmsb, tlsb;
    int16_t temperature_brut;
    float temperature;
    // Envoyer la commande de lecture
    Wire.beginTransmission(_address);
    Wire.write(DS1621_Read_Temperature);
    Wire.endTransmission();
    // Demande de lecture de 2 octets
    Wire.requestFrom(_address, (uint8_t)2);
    if (Wire.available() == 2)
    {
        tmsb = Wire.read();
        tlsb = Wire.read();
    }
    else
    {
        Serial.println("Erreur : pas assez de données reçues !");
        return -404; // Valeur d'erreur
    }
    // Traitement des données
    temperature_brut = (int8_t)tmsb;// MSB est déjà en complément à deux
    temperature = (float)temperature_brut + (tlsb >> 7) * 0.5;
    // Ajout du 0.5°C si nécessaire
    return temperature;}
```

La fonction DS1621::getValue() lit et retourne la température mesurée par le capteur DS1621. Elle commence par envoyer la commande de lecture de température à l'adresse du capteur via le bus I²C. Ensuite, elle demande deux octets de données correspondant à la température brute. Si les deux octets sont bien reçus, ils sont stockés dans tmsb (poids fort) et tlsb (poids faible). Le MSB, déjà codé en complément à deux, est converti en valeur entière signée. Le LSB est utilisé pour ajouter 0.5 degré si le bit de poids fort de ce dernier est à 1. La température finale, en degré Celsius, est alors calculée et renvoyée. Si la lecture échoue (moins de deux octets reçus), un message d'erreur est affiché et la fonction retourne une valeur d'erreur -404.

### La fonction startConv :

```
void DS1621::startConv()
{
    Wire.beginTransmission(_address);
    Wire.write(DS1621_start_reg);
    Wire.endTransmission();
}
```

La fonction DS1621::startConv() permet de démarrer une conversion de température sur le capteur DS1621. Elle utilise le bus I²C pour envoyer une commande d'activation : elle commence par initier une transmission vers l'adresse du capteur, envoie ensuite la commande DS1621_start_reg (qui correspond au registre de démarrage de la conversion), puis termine la transmission. Cette opération informe le capteur de commencer à mesurer la température.

La fonction stopConv :

```
void DS1621::stopConv()
{
    Wire.beginTransmission(_address);
    Wire.write(DS1621_stop_reg);
    Wire.endTransmission();
    Serial.println("Arrêt de la conversion de température.");
}
```

La fonction DS1621::stopConv() arrête la conversion de température du capteur DS1621. Elle initie une transmission $I^2C$ vers l'adresse du capteur, envoie la commande DS1621_stop_reg correspondant à l'arrêt de la conversion, puis termine la transmission. Enfin, elle affiche sur le moniteur série le message « Arrêt de la conversion de température. » pour indiquer que la commande a été envoyée.

# III- Module Real Time Clock

## III-1. Prise en main du module RTC3231

Le module RTC3231 nous permet d'avoir l'heure, les minutes, les secondes, ainsi que la date. La pile au dos lui permet de conserver l'heure même lorsqu'il n'est plus alimenté par l'ESP32.

Le module RTC3231 se présente sous la forme suivante :

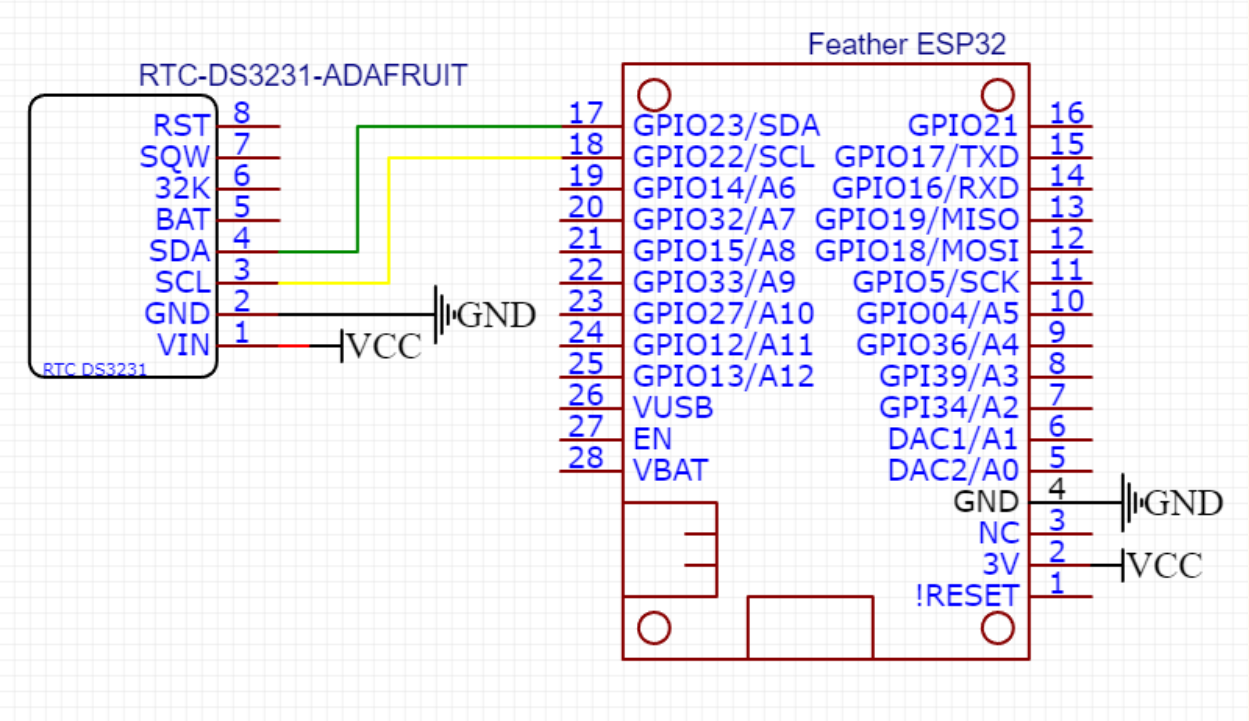

L'adresse I²C de ce module est configurée de la manière suivante :

| Data | A6 | A5 | A4 | A3 | A2 | A1 | A0 | Adresse |
|---|---|---|---|---|---|---|---|---|
| Binaire | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0b1101000 |
| Hexadécimal | 6 | | | 8 | | | 0 | 0x68 |

Ce module est muni de différents registres comme décrit dans le tableau ci-dessous

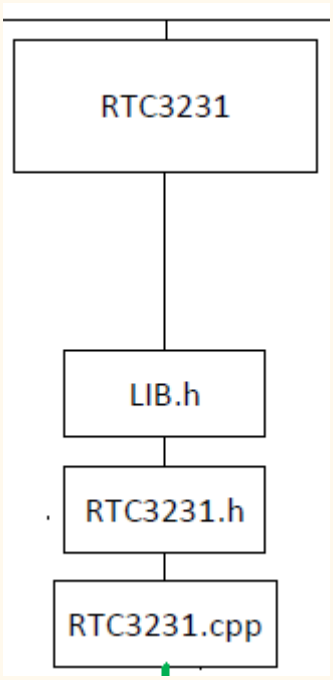
| Registre | Description |
|---|---|
| 0x00 | Contient les secondes en BCD (0 à 59) |
| 0x01 | Contient les minutes en BCD (0 à 59) |
| 0x02 | Contient les heures en BCD (format 12 ou 24 ) |
| 0x03 | Contient les jours (1 à 7) |
| 0x04 | Contient le jour du mois (1 à 31) |
| 0x05 | Contient les mois (1 à 12) |
| 0x06 | Contient l'année (0 à 99) |

Le module sera relié à l'ESP32 de la manière suivante :



## III-2. Création de la bibliothèque RTC3231

La bibliothèque RTC3231 est organisée de la manière suivante :




La bibliothèque LIB.h est la même que dans la partie II-2.

RTC3231.h nous permet d'initialiser notre classe RTC3231 et d'y intégrer les différentes méthodes de notre programme.

### A. RTC2331.h

```
#ifndef RTC_3231_H
#define RTC_3231_H
#include <LIB.h>
// les registres
#define RTC3231_seconds 0x00
#define RTC3231_minutes 0x01
#define RTC3231_hours 0x02
#define RTC3231_day 0x03
#define RTC3231_date 0x04
#define RTC3231_month 0x05
#define RTC3231_year 0x06
class RTC3231{
public:
    RTC3231();
    // les méthodes
    bool begin(uint8_t RTC3231_ADR);
    void setTime(uint8_t hh, uint8_t mm, uint8_t ss);
    void setDate(uint8_t index_day, uint8_t num, uint8_t index_month,
uint16_t year);
    uint8_t getHour();       // hh
    uint8_t getMinute();     // mm
    uint8_t getSeconds();    // ss
    String getStringDay();   // Dim, lun, Mar, Mer, Jeu, Ven,
    String getStringMonth(); // janv fev mars avril
    uint8_t getNumber();
    uint16_t getYear();     // 1 to 31
    String getStringTime(); // hh:mm
    String getStringDate(); // Day number month
    String getAllData();
private:
    uint8_t _address;
    uint8_t _binToBcd(uint8_t val);
    uint8_t _bcdToBin(uint8_t val);};
#endif
```

### B. RTC3231.cpp

Il contient l'implémentation en C++ des fonctions nécessaires à la communication avec le module RTC (Real Time Clock) DS3231 via le bus I²C. Ce module permet de maintenir une mesure précise de l'heure et de la date, même en cas de coupure d'alimentation, grâce à une pile de sauvegarde. Le code permet notamment de lire l'heure et la date courantes à partir des registres internes (secondes, minutes, heures, jour, date, mois, année) et de configurer ces valeurs. Ce fichier agit donc comme une interface entre le microcontrôleur et le composant DS3231, en masquant les détails bas niveau liés aux registres I²C, afin de simplifier son utilisation dans l'application principale.

La fonction begin :

```
bool RTC3231::begin(uint8_t RTC3231_adr)
{
    _address = RTC3231_adr;

    Wire.begin();
    Wire.beginTransmission(_address);
    if (Wire.endTransmission() == 0)
    {
        Serial.println("Capteur DS3231 connecté.");
        return true;
    }
    else
    {
        Serial.println("Échec de connexion au capteur DS3231 !");
        return false;
    }
}
```

La fonction RTC3231::begin(uint8_t RTC3231_adr) initialise la communication I²C avec le module RTC DS3231. Elle commence par stocker l'adresse I²C du module dans l'attribut _address de la classe. Ensuite, elle initialise le bus I²C avec Wire.begin(), puis tente une transmission vers

l'adresse spécifiée à l'aide de Wire.beginTransmission(_address) suivie de Wire.endTransmission(). Si cette transmission se termine sans erreur (valeur de retour égale à 0), cela signifie que le module DS3231 est bien connecté ; un message de confirmation est affiché dans le moniteur série et la fonction retourne true. Dans le cas contraire, un message d'erreur est affiché et la fonction retourne false.

La fonction _binToBcd :

```
uint8_t RTC3231::_binToBcd(uint8_t val)
{
    uint8_t A = val / 10;
    A = A * 16;
    uint8_t B = val % 10;
    uint8_t S = A + B;
    return S;
}
```

La fonction RTC3231::_binToBcd(uint8_t val) convertit une valeur en binaire vers le format BCD (Binary Coded Decimal), qui est utilisé par le module DS3231. Elle commence par diviser la valeur par 10 pour obtenir les dizaines, puis elle les multiplie par 16 pour les placer dans la partie haute du BCD. Ensuite, elle calcule les unités avec val % 10, puis additionne les deux parties. Par exemple, si la valeur en entrée est 45, la fonction renvoie 0x45 en BCD.

La fonction _BcdToBin :

```
uint8_t RTC3231::_bcdToBin(uint8_t val)
```

```
{
    uint8_t A = val / 16;
    A = A * 10;
    uint8_t B = val % 16;
    uint8_t S = A + B;
    return S;
}
```

La fonction RTC3231::_bcdToBin(uint8_t val) convertit une valeur en BCD (Binary Coded Decimal) en valeur binaire normale. Elle commence par diviser la valeur par 16 pour récupérer les dizaines, puis les multiplie par 10. Ensuite, elle récupère les unités avec val % 16. Enfin, elle additionne les dizaines et les unités pour obtenir la valeur finale en binaire. Par exemple, si val est égal à 0x45 en BCD, la fonction retournera 45 en binaire.

La fonction getSeconds :

```
uint8_t RTC3231::getSeconds()
{
    uint8_t seconds = 0;
    Wire.beginTransmission(_address);
    Wire.write(RTC3231_seconds);
    Wire.endTransmission();

    Wire.requestFrom(_address, byte(1));
    seconds = Wire.read();
    Wire.endTransmission();
    seconds = seconds & 0x7F;
    seconds = _bcdToBin(seconds);
    return seconds;
}
```

La fonction RTC3231::getSeconds() lit la valeur des secondes depuis le registre du module RTC DS3231. Elle commence par lancer une transmission I²C vers l'adresse du module, écrit l'adresse du registre des secondes, puis termine la transmission. Ensuite, elle demande un octet de données, lit cette valeur et termine la communication. Elle applique un masque binaire avec 0x7F pour ignorer le bit de contrôle, puis convertit le résultat du format BCD en binaire grâce à la fonction _bcdToBin(). Enfin, elle retourne la valeur des secondes sous forme entière normale.

La fonction getHour :

```
uint8_t RTC3231::getHour()
{
    uint8_t hours = 0;
    Wire.beginTransmission(_address);
    Wire.write(RTC3231_hours);
    Wire.endTransmission();
    Wire.requestFrom(_address, byte(1));
    hours = Wire.read();
    Wire.endTransmission();
    hours = hours & 0x7F;
    hours = _bcdToBin(hours);
    return hours;
}
```

La fonction RTC3231::getHour() lit la valeur des heures depuis le registre du module RTC DS3231. Elle commence par lancer une transmission I²C vers l'adresse du module, écrit l'adresse du registre des heures, puis termine la transmission. Ensuite, elle demande un octet de données, lit cette valeur et termine la communication. Elle applique un masque binaire avec 0x7F pour ignorer le bit de contrôle, puis convertit le résultat du format BCD en binaire grâce à la fonction _bcdToBin(). Enfin, elle retourne la valeur des heures sous forme entière normale.

La fonction getMinute :

```
uint8_t RTC3231::getMinute()
{
    uint8_t Minutes = 0;
    //
    Wire.beginTransmission(_address);
    Wire.write(RTC3231_minutes);
    Wire.endTransmission();

    Wire.requestFrom(_address, byte(1));
    Minutes = Wire.read();
    Wire.endTransmission();
    Minutes = Minutes & 0x7F;
    Minutes = _bcdToBin(Minutes);
    return Minutes;
}
```

La fonction RTC3231::getMinute() lit la valeur des minutes depuis le registre du module RTC DS3231. Elle commence par initier une transmission I²C vers l'adresse du module, écrit l'adresse du registre des minutes, puis termine la transmission. Ensuite, elle demande un octet de données, lit cette valeur et termine la communication. Elle applique un masque binaire avec 0x7F pour ignorer le bit de contrôle, puis convertit la valeur du format BCD en binaire grâce à la fonction _bcdToBin(). Enfin, elle retourne la valeur des minutes sous forme entière normale.

La fonction getStringDay :

```
String RTC3231::getStringDay()
{
    int day_ = 0;
    Wire.beginTransmission(_address);
    Wire.write(RTC3231_day);
    Wire.endTransmission();
    Wire.requestFrom(_address, byte(1));
    day_ = Wire.read();
    Wire.endTransmission();
    String day = days_week[day_];
    return day;
}
```

La fonction RTC3231::getStringDay() lit le jour de la semaine depuis le registre correspondant du module RTC DS3231. Elle commence par initier une transmission $I^2C$ vers l'adresse du module, écrit l'adresse du registre du jour, puis termine la transmission. Ensuite, elle demande un octet de données, lit cette valeur et termine la communication. Cette valeur est utilisée comme index pour accéder à un tableau days_week contenant les noms des jours de la semaine en texte. La fonction retourne alors ce nom sous forme de chaîne de caractères.

La fonction getStringMonth :

```
String RTC3231::getStringMonth()
{
    int Months_ = 0;
    Wire.beginTransmission(_address);
    Wire.write(RTC3231_month);
    Wire.endTransmission();
    Wire.requestFrom(_address, byte(1));
    Months_ = Wire.read();
    Wire.endTransmission();
    String Months = months_year[Months_];
    return Months;
}
```

La fonction RTC3231::getStringMonth() lit le mois depuis le registre correspondant du module RTC DS3231. Elle commence par initier une transmission I²C vers l'adresse du module, écrit l'adresse du registre du mois, puis termine la transmission. Ensuite, elle demande un octet de données, lit cette valeur et termine la communication. Cette valeur est utilisée comme index pour accéder à un tableau months_year contenant les noms des mois en texte. La fonction retourne alors ce nom sous forme de chaîne de caractères.

La fonction getNumber :

```
uint8_t RTC3231::getNumber()
{
    uint8_t Number = 0;
    Wire.beginTransmission(_address);
    Wire.write(RTC3231_date);
    Wire.endTransmission();
    Wire.requestFrom(_address, byte(1));
    Number = Wire.read();
    Wire.endTransmission();
    Number = Number & 0x7F;
    Number = _bcdToBin(Number);
    return Number;
}
```

La fonction RTC3231::getNumber() lit le jour du mois (la date) depuis le registre correspondant du module RTC DS3231. Elle commence par initier une transmission $I^2C$ vers l'adresse du module, écrit l'adresse du registre de la date, puis termine la transmission. Ensuite, elle demande un octet de données, lit cette valeur et termine la communication. Elle applique un masque binaire avec 0x7F pour ignorer le bit de contrôle, puis convertit la valeur du format BCD en binaire avec la fonction _bcdToBin(). Enfin, elle retourne la valeur du jour du mois sous forme entière normale.

La fonction getYear :

```
uint16_t RTC3231::getYear()
{
    Wire.beginTransmission(_address);
    Wire.write(RTC3231_year); // Adresse du registre de l'année
    Wire.endTransmission();
    Wire.requestFrom(_address, byte(1));
    Wire.endTransmission();
    uint8_t year = Wire.read();    // Lire l'année en BCD
    return _bcdToBin(year) + 2000; // Convertir et ajouter 2000
}
```

La fonction RTC3231::getYear() lit l'année depuis le registre dédié du module RTC DS3231. Elle commence par initier une transmission I²C vers l'adresse du module, écrit l'adresse du registre de l'année, puis termine la transmission. Ensuite, elle demande un octet de données, termine la communication, puis lit la valeur reçue (année en format BCD). Elle convertit cette valeur en binaire avec la fonction _bcdToBin() puis ajoute 2000 pour obtenir l'année complète (exemple : 23 devient 2023). Enfin, elle retourne cette valeur sous forme entière.

La fonction getStringTime :

```
String RTC3231::getStringTime()
{
    uint8_t hours = getHour();
    uint8_t minutes = getMinute();
    uint8_t seconds = getSeconds();
    String hoursStr = String(hours);
    String minutesStr = String(minutes);
    String secondsStr = String(seconds);
    if (hours < 10)
        hoursStr = "0" + hoursStr; // Ajoute un zéro devant l'heure si <
10
    if (minutes < 10)
        minutesStr = "0" + minutesStr; // Ajoute un zéro devant les
minutes si < 10
    if (seconds < 10)
        secondsStr = "0" + secondsStr; // Ajoute un zéro devant les
secondes si < 10
    String Heures_assemblee = hoursStr + ":" + minutesStr + ":" +
secondsStr;
    return Heures_assemblee;
}
```

La fonction RTC3231::getStringTime() récupère l'heure, les minutes et les secondes en appelant respectivement les fonctions getHour(), getMinute() et getSeconds(). Elle convertit ensuite ces valeurs en chaînes de caractères. Pour chaque valeur inférieure à 10, elle ajoute un zéro devant pour garantir un format à deux chiffres. Enfin, elle assemble les heures, minutes et secondes au format "HH:MM:SS" et retourne cette chaîne de caractères représentant l'heure actuelle.

La fonction getStringDate :

```
String RTC3231::getStringDate()
{
    String Number = String((uint8_t)getNumber());
    String Day = getStringDay();
    String Month = getStringMonth();
    String Date_complete = Day + " " + Number + " " + Month;
    return Date_complete;
}
```

La fonction RTC3231::getStringDate() récupère le jour du mois avec getNumber(), le nom du jour de la semaine avec getStringDay() et le nom du mois avec getStringMonth(). Elle convertit le jour du mois en chaîne de caractères, puis assemble ces informations dans une seule chaîne au format "Jour NomJour NuméroJour NomMois". Enfin, elle retourne cette chaîne représentant la date complète sous forme lisible.

La fonction getAllData :

```
String RTC3231::getAllData()
{
    String Heures_ = getStringTime();
    String Dates_ = getStringDate();
    String Annee_ = String((uint16_t)getYear());
    String allData = Dates_ + " " + Annee_ + " " + Heures_;
    Serial.println(allData);
    return allData;
}
```

La fonction RTC3231::getAllData() récupère l'heure complète avec getStringTime(), la date complète avec getStringDate(), ainsi que l'année avec getYear(). Elle convertit l'année en chaîne de caractères, puis assemble toutes ces informations dans une seule chaîne au format "Jour NomJour NuméroJour NomMois Année HH:MM:SS". Cette chaîne est affichée dans le moniteur série avec Serial.println() puis retournée par la fonction.

La fonction setTime :

```
void RTC3231::setTime(uint8_t hh, uint8_t mm, uint8_t ss)
{
    Wire.beginTransmission(_address);
    Wire.write(0x00); // Adresse du registre de l'heure
    Wire.write(_binToBcd(ss));
    Wire.write(_binToBcd(mm));
    Wire.write(_binToBcd(hh));
    Wire.endTransmission();
}
```

La fonction RTC3231::setTime(uint8_t hh, uint8_t mm, uint8_t ss) permet de régler l'heure du module RTC DS3231. Elle commence une transmission $I^2C$ vers l'adresse du module, puis écrit l'adresse du registre de l'heure (0x00). Ensuite, elle écrit les secondes, minutes et heures, après les avoir converties du format binaire en format BCD grâce à la fonction _binToBcd(). Enfin, elle termine la transmission, ce qui met à jour l'heure du module.

La fonction setDate :

```
void RTC3231::setDate(uint8_t index_day, uint8_t num, uint8_t
index_month, uint16_t year){
    Wire.beginTransmission(_address);
    Wire.write(RTC3231_day);// Adresse du registre du jour de la semaine
    Wire.write(_binToBcd(index_day));// Jour de la semaine (1 = Lundi,
..., 7 = Dimanche)
    Wire.write(_binToBcd(num));          // Jour du mois (1-31)
    Wire.write(_binToBcd(index_month)); // Mois (1-12)
    Wire.write(_binToBcd(year - 2000)); // Année (0-99)
    Wire.endTransmission();}
```

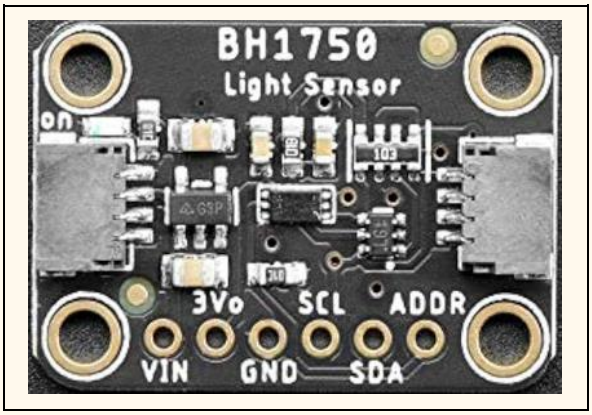
La fonction RTC3231::setDate(uint8_t index_day, uint8_t num, uint8_t index_month, uint16_t year) permet de configurer la date sur le module RTC DS3231. Elle commence une transmission $I^2C$ vers l'adresse du module, puis écrit l'adresse du registre du jour de la semaine (RTC3231_day). Ensuite, elle envoie successivement le jour de la semaine, le jour du mois, le mois et l'année, chacun converti du format binaire en BCD via la fonction _binToBcd(). L'année est ajustée en soustrayant 2000 pour correspondre au format du DS3231 (valeurs entre 0 et 99). Enfin, elle termine la transmission pour mettre à jour la date dans le module.

# IV- Capteur de luminosité

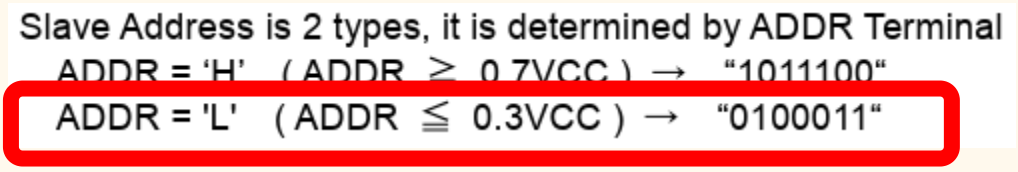
## IV-1. Prise en main du capteur BH1750

Pour le capteur de luminosité nous allons utiliser le capteur BH1750. Le rôle de ce capteur est de mesurer la luminosité ambiante.

Le capteur prendra la forme d'une plaquette comme ci-dessous :




L'adresse I²C de ce capteur est configuré de la manière suivante :

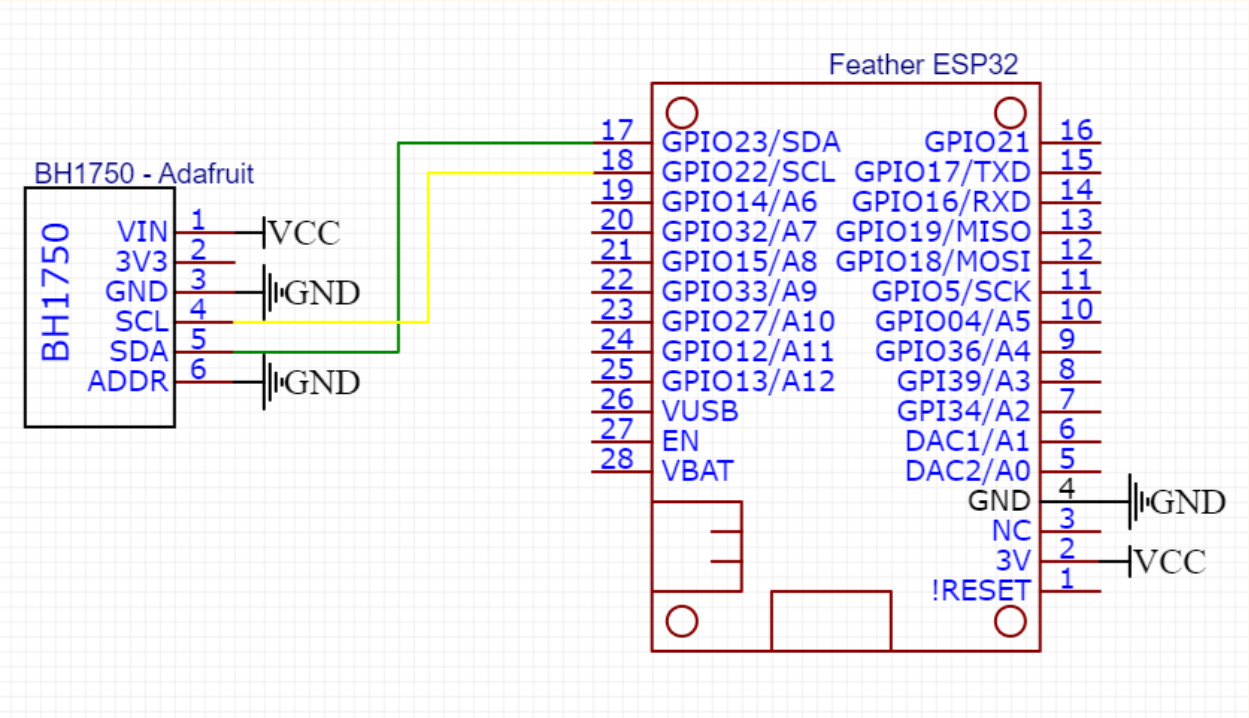



| Data | A6 | A5 | A4 | A3 | A2 | A1 | A0 | Adresse |
|---|---|---|---|---|---|---|---|---|
| Binaire | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0b0100011 |
| Hexadécimal | 0 | 2 | | 0 | | 2 | 1 | 0x23 |

Ce capteur est muni de différents registres comme décrit dans le tableau ci-dessous

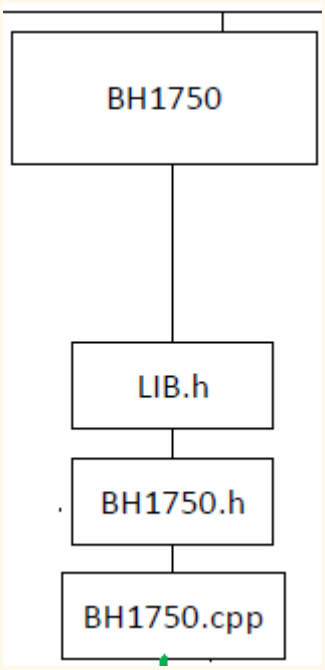
| Registre | Description |
|---|---|
| 0x00 | Met le capteur en mode veille (économie d'énergie). |
| 0x01 | Active le capteur (mais sans commencer de mesure). |
| 0x10 | Mesure continue, haute résolution (1 lx précision, 120 ms). |
| 0x11 | Mesure continue, haute résolution (1 lx, mais lecture plus lente). |
| 0x13 | Mesure continue, basse résolution (4 lx précision, 16 ms). |
| 0x20 | Mesure unique, haute résolution (capteur éteint après mesure). |
| 0x21 | Mesure unique, haute résolution 2 |
| 0x23 | Mesure unique, basse résolution |

Le capteur sera relié à l'ESP32 de la manière suivante :



## IV-2. Création de la bibliothèque BH1750

La bibliothèque BH1750 est organisée de la manière suivante :




La bibliothèque LIB.h est la même que dans la partie II-2.

BH1750.h nous permet d'initialiser notre classe BH1750 et d'y intégrer les différentes méthodes de notre programme.

## A. BH1750.h

```
#ifndef BH_1750_H
#define BH_1750_H

#include <LIB.h>
#define BH1750_POWER_DOWN 0x00
#define BH1750_POWER_ON 0x01
#define BH1750_CONTINOUS_H_RES_MOD 0x10
#define BH1750_CONTINOUS_H_RES_MOD_2 0x11
#define BH1750_CONTINOUS_L_RES_MOD 0x13
#define BH1750_ONE_TIME_H_RES_MOD 0x20
#define BH1750_ONE_TIME_H_RES_MOD_2 0x21
#define BH1750_ONE_TIME_L_RES_MOD 0x23
class BH1750
{
public:
    // constructor
    BH1750();
    bool begin(uint8_t BH1750_ADR);
    void setMode(uint8_t mode);
    bool isMeasurementReady();
    void powerOn();
    void powerDown();
    float readValue();
    String luminosite_directe();
private:
    uint8_t _address;
    float _value;
    uint8_t _mode;
    uint32_t _lastTime;
};
#endif
```

## B. BH1750.cpp

Il regroupe l'ensemble du code nécessaire à la communication avec le capteur de luminosité BH1750 via le bus I²C. Ce capteur est capable de mesurer l'intensité lumineuse ambiante exprimée en lux, avec une précision allant jusqu'à 1 lx, selon le mode choisi.

Ce fichier contient les fonctions permettant :

- d'initialiser le capteur,
- de configurer son mode de fonctionnement (mesure continue ou ponctuelle, haute ou basse résolution),
- de récupérer et convertir les données de luminosité.

### La fonction begin :

```
bool BH1750::begin(uint8_t address)
{
    _address = address;
    Wire.begin();
    Wire.beginTransmission(_address);
    if (Wire.endTransmission() == 0)
    {
        Serial.println("Capteur BH1750 connecté.");
        return true;
    }
    else
    {
        Serial.println("Échec de connexion au capteur BH1750 !");
        return false;
    }
}
```

La fonction BH1750::begin initialise la communication I²C avec le capteur BH1750. Elle stocke l'adresse du capteur, démarre le bus I²C, puis tente de contacter le capteur. Si le capteur répond correctement, la fonction affiche un message indiquant que le capteur est connecté et retourne true. Sinon, elle affiche un message d'erreur et retourne false.

La fonction setMode :

```
void BH1750::setMode(uint8_t mode)
{
    _mode = mode;
    Wire.beginTransmission(_address);
    Wire.write(_mode);
    Wire.endTransmission();
}
```

La fonction BH1750::setMode configure le mode de fonctionnement du capteur BH1750. Elle stocke le mode choisi dans la variable _mode, puis envoie cette commande au capteur via le bus $I^2C$. Cela permet de sélectionner le type de mesure (continue ou ponctuelle) et la résolution (haute ou basse) selon le mode défini.

La fonction isMeasurmentReady :

```
bool BH1750::isMeasurementReady() {
    uint32_t timedelay = 0;
    bool res = false;
    uint32_t now = 0;
    switch (_mode)
    {
    case BH1750_CONTINOUS_H_RES_MOD:
        timedelay = 180;
        break;
    case BH1750_CONTINOUS_H_RES_MOD_2:
        timedelay = 180;
        break;
    case BH1750_CONTINOUS_L_RES_MOD:
        timedelay = 180;
        break;
    case BH1750_ONE_TIME_H_RES_MOD:
        timedelay = 24;
        break;
    case BH1750_ONE_TIME_H_RES_MOD_2:
        timedelay = 24;
        break;
    case BH1750_ONE_TIME_L_RES_MOD:
        timedelay = 24;
        break;
    default:
        timedelay = 180;
        break;
    }
    now = millis();
    if ((now - _lastTime) > timedelay)
        res = true;
    return res;
}
```

La fonction BH1750::isMeasurementReady vérifie si le temps nécessaire pour que la mesure soit prête est écoulé selon le mode de fonctionnement actuel du capteur. Elle définit un délai en millisecondes adapté au mode utilisé, puis compare ce délai au temps écoulé depuis la dernière mesure (_lastTime). Si ce temps est dépassé, la fonction retourne true indiquant que la mesure est prête, sinon elle retourne false.

La fonction powerOn :

```
void BH1750::powerOn()
{
    Wire.beginTransmission(_address);
    Wire.write(BH1750_POWER_ON);
    Wire.endTransmission();
}
```

La fonction BH1750::powerOn envoie une commande au capteur BH1750 pour l'allumer. Elle utilise la communication $I^2C$ pour transmettre le code de mise sous tension au capteur, ce qui le prépare à effectuer des mesures de luminosité.

La fonction powerDown :

```
void BH1750::powerDown()
{
    Wire.beginTransmission(_address);
    Wire.write(BH1750_POWER_DOWN);
    Wire.endTransmission();
}
```

La fonction BH1750::powerDown envoie une commande au capteur BH1750 pour le mettre en mode veille. Elle utilise la communication $I^2C$ pour transmettre le code d'arrêt au capteur, ce qui réduit sa consommation d'énergie.

La fonction readValue :

```
float BH1750::readValue()
{
    uint16_t data = 0x0000;
    uint8_t upper = 0x00;
    uint8_t lower = 0x00;
    float res = 0;

    Wire.requestFrom(_address, byte(2));
    upper = Wire.read();
    lower = Wire.read();
    Wire.endTransmission();

    data = (upper << 8 | lower);
    res = data / 1.20f;
    _lastTime = millis();
    return res;
}
```

La fonction BH1750::readValue lit la valeur de luminosité mesurée par le capteur BH1750. Elle demande 2 octets de données via $I^2C$, récupère les octets supérieurs et inférieurs, puis les combine pour former une valeur brute. Cette valeur est convertie en lux en la divisant par 1,20. La fonction met à jour le temps de la dernière mesure et retourne la luminosité en lux.

### La fonction luminosité directe :

```
String BH1750::luminosite_directe()
{
    String lux = "0";
    if (isMeasurementReady())
    {
        lux = readValue();
    }
    return lux;
}
```

La fonction BH1750::luminosite_directe vérifie si une mesure est prête en appelant isMeasurementReady. Si c'est le cas, elle lit la valeur de luminosité avec readValue et la retourne sous forme de chaîne de caractères. Sinon, elle retourne "0".

# V- Afficheur OLED

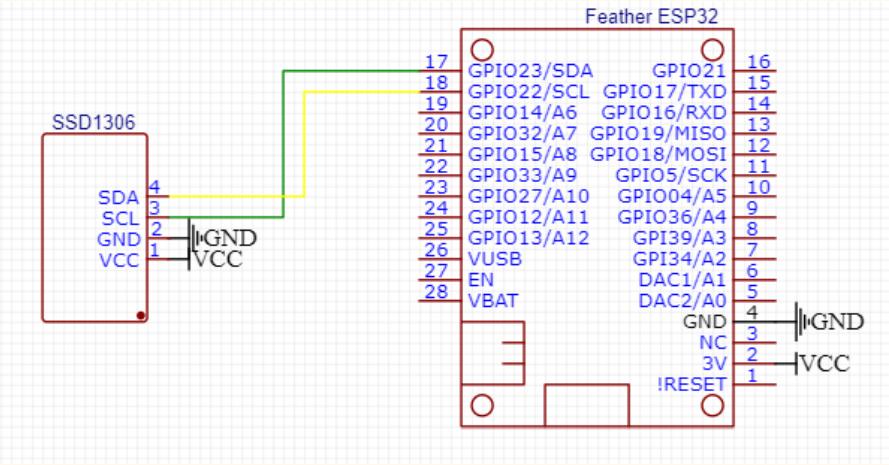
## V-1. Prise en main de l'afficheur OLED SSD1306

Pour l'affichage de notre projet nous avons choisi un écran monochrome OLED le SSD1306, il a une définition de 128 x 64 pixels.

Son adresse $I^2C$ est fixée à 0x3C.

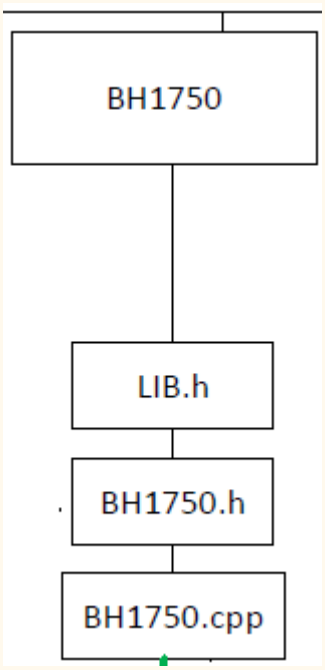
Il prend la forme d'une petite plaquette comme ci-dessous :

Il sera intégré et connecté à l'ESP32 de la manière suivante :



## V-2. Création de la bibliothèque SSD1306

La bibliothèque SSD1306 est organisé de la manière suivante :




La bibliothèque LIB.h est la même que dans la partie II-2.

Le fichier SSD1306.h définit une classe permettant de gérer un écran OLED basé sur le contrôleur SSD1306. Il inclut les bibliothèques nécessaires et déclare des objets pour interagir avec plusieurs capteurs : RTC3231 (horloge temps réel), DS1621 (température), BH1750 (luminosité) et AHT20 (température et humidité). La classe SSD1306 fournit plusieurs méthodes. Cette classe simplifie la gestion de l'affichage des différentes données mesurées sur un écran OLED.

## A. SSD1306.h

```
#ifndef SSD1306_H
#define SSD1306_H
#include <LIB.h>
#include <OBJ.h>
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
extern Adafruit_SSD1306 oled;
extern RTC3231 myRTC3231;
extern DS1621 myDS1621;
extern BH1750 myBH1750;
extern AHT20 myAHT20;
class SSD1306
{
public:
    // constructeur
    SSD1306();
    void beginOLED(uint8_t SSD1306_ADR);
    void affichage_heures();
    void affichage_menus(int i);
    String affichage_temperature();
    String returned_temperature();
    void affichage_droit_auteur();
    void affichage_luminosite();
    void affichage_humidite();
private:
    uint8_t _address;
    int _value;
};
#endif
```

### B. images.h

Il contient tous les bitmaps utilisés pour l'affichage graphique. Chaque image ou icône est représentée sous forme de tableau de valeurs correspondant aux pixels. Plutôt que de détailler les centaines de lignes de données brutes, il faut simplement retenir qu'un tableau principal, epd_bitmap_allArray, regroupe l'ensemble des pointeurs vers ces images.

Ce tableau permet d'accéder facilement à n'importe quelle icône ou image en fonction de son index, ce qui simplifie leur utilisation dans le programme principal pour l'affichage à l'écran.

### C. SSD1306.cpp

Il gère l'affichage des données sur un écran OLED piloté par le contrôleur SSD1306 en utilisant la bibliothèque Adafruit_SSD1306. Il centralise les fonctions permettant d'afficher l'heure, la date, la température, la luminosité et l'humidité, en s'appuyant sur différents capteurs (RTC3231, DS1621, BH1750, AHT20). La fonction beginOLED() initialise l'écran, tandis que d'autres fonctions comme affichage_heures(), affichage_temperature(), affichage_luminosite() ou encore affichage_humidite() se chargent respectivement d'afficher les valeurs mesurées. Le fichier permet aussi d'afficher des icônes via la fonction affichage_menus(int i) et de présenter un écran de droits d'auteur avec affichage_droit_auteur(). L'ensemble assure une interface visuelle complète et dynamique.

La fonction beginOLED :

```
void SSD1306::beginOLED(uint8_t SSD1306_adr)
{
    _address = SSD1306_adr;
    if (!oled.begin(SSD1306_SWITCHCAPVCC, _address))
    {
        Serial.println(F("Échec de l'initialisation de l'écran OLED"));
        while (true);}
    else
    {
        Serial.println("Capteur SSD1306 connecté");
    }
    oled.clearDisplay();
    oled.display();
}
```

Cette fonction initialise l'écran OLED avec l'adresse fournie. Si l'initialisation échoue, un message d'erreur est affiché et le programme s'arrête dans une boucle infinie. Sinon, elle confirme la connexion du capteur OLED, puis nettoie l'écran avant d'afficher un écran vide.

La fonction affichage_heures :

```
void SSD1306::affichage_heures()
{
    oled.clearDisplay();
    oled.drawRect(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, WHITE);
    oled.drawBitmap(5, 5, epd_bitmap_allArray[ICON_DATE_HEURE], 32, 32,
WHITE);
    oled.setTextColor(WHITE);
    oled.setTextSize(2);
    oled.setCursor(50, 5);
    oled.print(myRTC3231.getStringDay());
    oled.setCursor(94, 5);
    oled.print(myRTC3231.getNumber());
    oled.setCursor(50, 25);
    oled.print(myRTC3231.getStringMonth());
    oled.setCursor(20, 45);
    oled.print(myRTC3231.getStringTime());
    oled.display();
    delay(TEMPO);
    return;
}
```

Cette fonction efface l'écran OLED, dessine un cadre blanc autour de tout l'écran, affiche une icône de date/heure, puis affiche le jour de la semaine, le numéro du jour, le mois, et enfin l'heure complète récupérés via le module RTC3231. Elle actualise ensuite l'affichage et marque une pause avec delay pour laisser le temps à l'utilisateur de voir l'information.

La fonction affichage_menus :

```
void SSD1306::affichage_menus(int i)
{
    oled.clearDisplay();
    oled.drawBitmap(0, 0, epd_bitmap_allArray[i], 128, 64, WHITE);
    oled.display();
}
```

Cette fonction efface l'écran OLED puis affiche une image (bitmap) choisie dans le tableau epd_bitmap_allArray selon l'indice i passé en paramètre. Enfin, elle met à jour l'écran pour afficher l'image sélectionnée.

La fonction affichage_temperature :

```
String SSD1306::affichage_temperature()
{
    oled.clearDisplay();
    oled.drawRect(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, WHITE);
    oled.drawBitmap(4, 14, epd_bitmap_allArray[ICON_TEMPERATURE], 32,
32, WHITE);
    oled.setTextColor(WHITE);
    oled.setTextSize(2);
    oled.setCursor(41, 23);
    float temperature_moyenne = (myDS1621.getValue() +
myAHT20.temperature()) / 2;
    String temperature_returned = String((float)temperature_moyenne);
    oled.print(temperature_moyenne);
    oled.print((char)247);
    oled.print("C");
    oled.display();
    delay(TEMPO);
    return temperature_returned;
}
```

Cette fonction efface l'écran OLED, dessine un cadre, affiche une icône de température, calcule la moyenne des températures mesurées par deux capteurs (DS1621 et AHT20), affiche cette température moyenne avec le symbole du degré Celsius, puis actualise l'écran. Elle renvoie aussi cette température sous forme de chaîne de caractères (String).

### La fonction returned_temperature :

```
String SSD1306::returned_temperature()
{
    float temperature_moyenne = (myDS1621.getValue() +
myAHT20.temperature()) / 2;
    String temperature_returned = String((float)temperature_moyenne);
    return temperature_returned;
}
```

Cette fonction calcule la température moyenne issue des deux capteurs DS1621 et AHT20, convertit cette valeur en chaîne de caractères (String) et la retourne sans afficher quoi que ce soit sur l'écran.

La fonction affichage_droit_auteur :

```
void SSD1306::affichage_droit_auteur()
{
    oled.clearDisplay();
    oled.setTextColor(WHITE);
    oled.setTextSize(2);
    oled.drawBitmap(0, 0, epd_bitmap_allArray[DROIT_CY], 128, 64,
WHITE);
    oled.display();
    delay(5000);
    oled.clearDisplay();
    oled.drawBitmap(48, 0, epd_bitmap_allArray[DROIT_AUTEUR], 32, 32,
WHITE);
    oled.setCursor(10, 30);
    oled.print("Lorenzo.S");
    oled.setCursor(10, 45);
    oled.print("Quentin.P");
    oled.display();
    delay(2000);
    return;
}
```

Cette fonction affiche d'abord un écran complet avec un bitmap représentant le logo de CYU IUT pendant 5 secondes, puis affiche un autre petit bitmap (logo copyright) avec les noms des auteurs Lorenzo S. et Quentin P. pendant 2 secondes.

### La fonction affichage_luminosite :

```
void SSD1306::affichage_luminosite()
{
    oled.clearDisplay();
    oled.drawRect(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, WHITE);
    oled.drawBitmap(48, 0, epd_bitmap_allArray[ICON_ILLUMINATION], 32,
32, WHITE);
    oled.setTextColor(WHITE);
    oled.setTextSize(2);
    oled.setCursor(10, 44);
    oled.print(myBH1750.readValue());
    oled.print(" Lx");
    oled.display();
    delay(10000);
    return;
}
```

La fonction affichage_luminosite efface l'écran, dessine un cadre et affiche une icône d'illumination en haut au centre. Ensuite, elle affiche la valeur mesurée de luminosité (en lux) en gros caractères vers le bas. L'écran est mis à jour, puis la fonction attend 10 secondes avant de se terminer.

La fonction affichage_humidite :

```
void SSD1306::affichage_humidite()
{
    oled.clearDisplay();
    oled.drawRect(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, WHITE);
    oled.drawBitmap(48, 0, epd_bitmap_allArray[ICON_HUMIDITE], 32, 32,
WHITE);
    oled.setTextColor(WHITE);
    oled.setTextSize(2);
    oled.setCursor(10, 44);
    oled.print(myAHT20.humidity());
    oled.print(" %");
    oled.display();
    delay(10000);
    return;
}
```

La fonction affichage_humidite efface l'écran, dessine un cadre, et affiche une icône représentant l'humidité en haut au centre. Ensuite, elle affiche la valeur d'humidité mesurée (en pourcentage) en gros caractères vers le bas. L'écran est mis à jour, puis la fonction attend 10 secondes avant de se terminer.

# VI- Capteur d'humidité et de température

## VI-1. Prise en main du capteur AHT20

Pour le capteur d'humidité nous allons utiliser le capteur AHT20. Le rôle de ce capteur est de mesurer l'humidité ambiante.

Le capteur prendra la forme d'une plaquette comme ci-dessous :

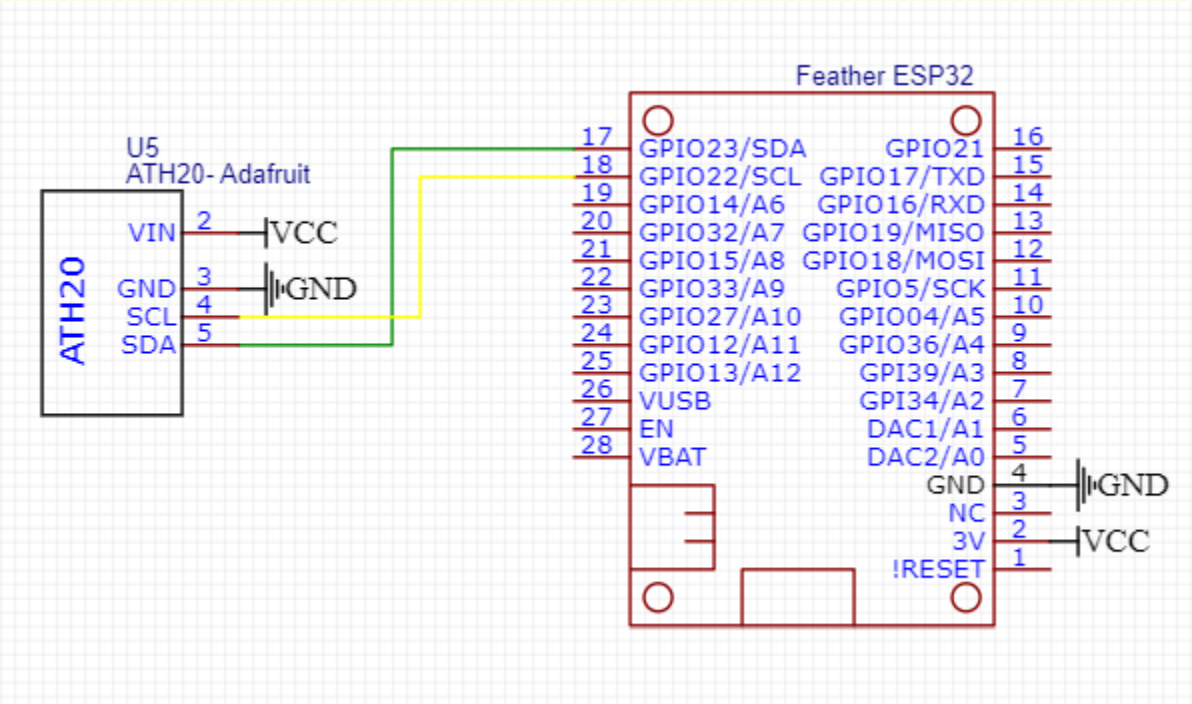

L'adresse I²C de ce capteur est configurée de la manière suivante :


| Data | A6 | A5 | A4 | A3 | A2 | A1 | A0 | Adresse |
|---|---|---|---|---|---|---|---|---|
| Binaire | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0b0111000 |
| Hexadécimal | 0 | 2 | 1 | 8 | | 0 | 0 | 0x38 |

Ce capteur est muni de différents registres comme décrit dans le tableau ci-dessous

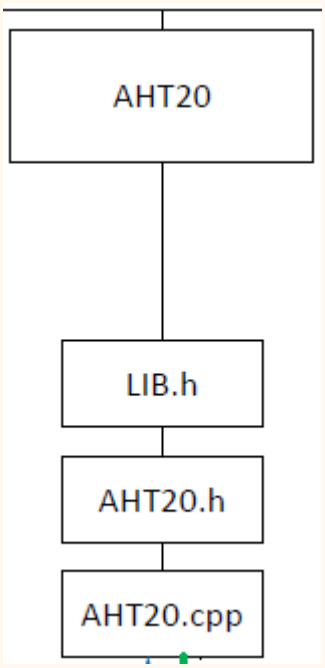
| Registre | Description |
|---|---|
| 0x00 | Deuxième paramètre de la commande d'initialisation |
| 0x00 | Deuxième paramètre pour déclencher la mesure |
| 0x08 | Premier paramètre de la commande d'initialisation |
| 0x33 | Premier paramètre pour déclencher la mesure |
| 0x71 | Commande pour lire le statut du capteur |
| 0xAC | Commande pour déclencher une mesure (température + humidité) |
| 0xBE | Commande d'initialisation du capteur |

Le capteur sera relié à l'ESP32 de la manière suivante :



## VI-2. Création de la bibliothèque AHT20

La bibliothèque AHT20 est organisée de la manière suivante :




La bibliothèque LIB.h est la même que dans la partie II-2.

Ce fichier d'en-tête déclare la classe AHT20 permettant de gérer le capteur de température et d'humidité AHT20 via le protocole $I^2C$. Il définit les constantes nécessaires à la communication avec le capteur (commandes, temps d'attente, paramètres d'initialisation et de mesure) et propose une interface simple pour initialiser le capteur et récupérer les mesures environnementales.

## A. AHT20.h

```
#ifndef AHT_20_H
#define AHT_20_H
#include <LIB.h>

#define AHT20_CMD_INIT_PARAMS_2ND 0x00
#define AHT20_CMD_MEASUREMENT_PARAMS_2ND 0x00
#define AHT20_CMD_INIT_PARAMS_1ST 0x08
#define AHT20_CMD_MEASUREMENT_PARAMS_1ST 0x33
#define AHT20_CMD_STATUS 0x71
#define AHT20_CMD_MEASUREMENT 0xAC
#define AHT20_CMD_INIT 0xBE
#define AHT20_CMD_MEASUREMENT_TIME 80
#define AHT20_CMD_POWER_ON_TIME 40
#define AHT20_CMD_INIT_TIME 10
class AHT20
{
public:
    AHT20();
    bool begin(uint8_t AHT20_ADR);
    bool getValues();
    float humidity();
    float temperature();

private:
    float _humidity;
    float _temperature;
    uint8_t _address;
    bool _start();
};
#endif
```

## B. AHT20.cpp

Il contient l'implémentation des fonctions permettant de communiquer avec le capteur AHT20 via le protocole I²C. Il permet d'initialiser le capteur, de lancer une mesure d'humidité et de température, puis de récupérer et convertir les données brutes reçues. La méthode begin établit la connexion et envoie les paramètres d'initialisation nécessaires. La fonction getValues déclenche une mesure et lit les données envoyées par le capteur après un temps d'attente. Les fonctions humidity et temperature retournent respectivement l'humidité relative et la température ambiante calculées à partir des données brutes selon les formules du constructeur. Ce fichier repose sur des constantes bien définies pour assurer une utilisation claire et conforme à la documentation du capteur.

La fonction begin :

```
bool AHT20::begin(uint8_t address)
{
    _address = address;
    bool state = false;
    uint8_t calibrationStatus = 0xFF;
    Wire.begin();
    delay(AHT20_CMD_POWER_ON_TIME);
    Wire.beginTransmission(_address);
    if (Wire.endTransmission() == 0)
    {
        Serial.println("Capteur AHT20 connecté.");
        Wire.beginTransmission(_address);
        Wire.write(AHT20_CMD_STATUS);
        Wire.endTransmission();
        Wire.requestFrom(_address, byte(1));
        calibrationStatus = Wire.read();
        Wire.endTransmission();
        if (calibrationStatus & 0b00001000 == 0)
        {
            Wire.beginTransmission(_address);
            Wire.write(AHT20_CMD_INIT);
            Wire.write(AHT20_CMD_INIT_PARAMS_1ST);
            Wire.write(AHT20_CMD_INIT_PARAMS_2ND);
            Wire.endTransmission();
            delay(AHT20_CMD_INIT_TIME);
        }
        Wire.beginTransmission(_address);
        Wire.write(0xBE);
        Wire.endTransmission();
        return true;
    }
    else
    {
        Serial.println("Échec de connexion au capteur AHT20 !");
        return false;
```

```
    }
    return state;
}
```

La fonction begin initialise la communication avec le capteur AHT20 en utilisant le protocole I²C. Elle commence par enregistrer l'adresse du capteur, initialise le bus I²C avec Wire.begin(), puis effectue un test de connexion. Si le capteur répond, elle vérifie l'état de calibration en lisant le registre de statut. Si le capteur n'est pas encore calibré, elle envoie la commande d'initialisation avec les paramètres requis. Une fois l'initialisation effectuée, la fonction renvoie true pour signaler que le capteur est prêt à être utilisé. En cas d'échec de connexion, elle affiche un message d'erreur et renvoie false.

La fonction _start :

```
bool AHT20::_start() {
    uint8_t isBusy = 0xFF;
    uint32_t now = 0;
    Wire.beginTransmission(_address);
    Wire.write(AHT20_CMD_MEASUREMENT);
    Wire.write(AHT20_CMD_MEASUREMENT_PARAMS_1ST);
    Wire.write(AHT20_CMD_MEASUREMENT_PARAMS_2ND);
    Wire.endTransmission();
    delay(80);
    now = millis();
    do{
        if (millis() - now > 200){
            return false;
        }
        Wire.beginTransmission(_address);
        Wire.write(AHT20_CMD_STATUS);
        Wire.endTransmission();
        Wire.requestFrom(_address, byte(1));
        isBusy = Wire.read();
        Wire.endTransmission();
        delay(1);
    } while (isBusy & 0x80);
    return true; }
```

La fonction privée _start du fichier AHT20.cpp déclenche une mesure de température et d'humidité sur le capteur AHT20. Elle commence par envoyer la commande de mesure accompagnée de ses paramètres via le bus $I^2C$. Ensuite, elle attend un délai de traitement de 80 millisecondes, puis vérifie si le capteur est toujours occupé en lisant le registre de statut. Cette vérification se fait en boucle avec une temporisation, jusqu'à ce que le bit d'occupation soit libéré ou qu'un délai maximal de 200 millisecondes soit dépassé. Si le capteur est prêt à transmettre les données, la fonction renvoie true, sinon elle retourne false pour indiquer une erreur ou un dépassement de temps.

La fonction getValues :

```
bool AHT20::getValues()
{
    bool res = true;
    uint8_t values[7] = {0, 0, 0, 0, 0, 0, 0};
    uint8_t index = 0;
    uint32_t temp = 0;
    res = _start();
    if (res == false)
    {
        return false;
    }
    Wire.requestFrom(_address, byte(7));
    while (Wire.available())
    {
        values[index] = Wire.read();
        index++;
    }
    Wire.endTransmission();
    temp = values[1];
    temp <<= 8;
    temp |= values[2];
    temp <<= 4;
    temp |= (values[3] >> 4);
    _humidity = (temp * 100.0f) / 1048576;
    temp = values[3] & 0x0F;
    temp <<= 8;
    temp |= values[4];
    temp <<= 8;
    temp |= values[5];
    _temperature = (temp * 200.0f) / 1048576 - 50;
    return true;
}
```

La fonction getValues permet d'effectuer une acquisition des valeurs mesurées par le capteur AHT20. Elle commence par appeler la fonction _start qui initie une mesure. Si cette étape échoue, la fonction retourne false. Ensuite, elle lit les sept octets de données envoyés par le capteur via le bus $I^2C$. Les trois premiers octets sont utilisés pour calculer l'humidité relative, codée sur 20 bits, en combinant les bits pertinents selon le protocole du capteur. Les trois octets suivants contiennent la température, également codée sur 20 bits. Les valeurs numériques brutes sont converties en pourcentage d'humidité et en degrés Celsius selon les formules fournies par le fabricant. Les résultats sont stockés dans les variables internes de la classe et la fonction re tourne true pour indiquer que la lecture a réussi.

### La fonction humidity :

```
float AHT20::humidity(){
    float humidity = 0.0f;
    getValues();
    humidity = _humidity;
    return humidity;
}
```

Cette fonction permet de récupérer la dernière valeur d'humidité mesurée par le capteur AHT20. Elle appelle d'abord la fonction getValues pour déclencher une nouvelle mesure et mettre à jour les données internes. La variable _humidity, contenant le pourcentage d'humidité calculé à partir des données brutes du capteur, est ensuite retournée sous forme d'un flottant.

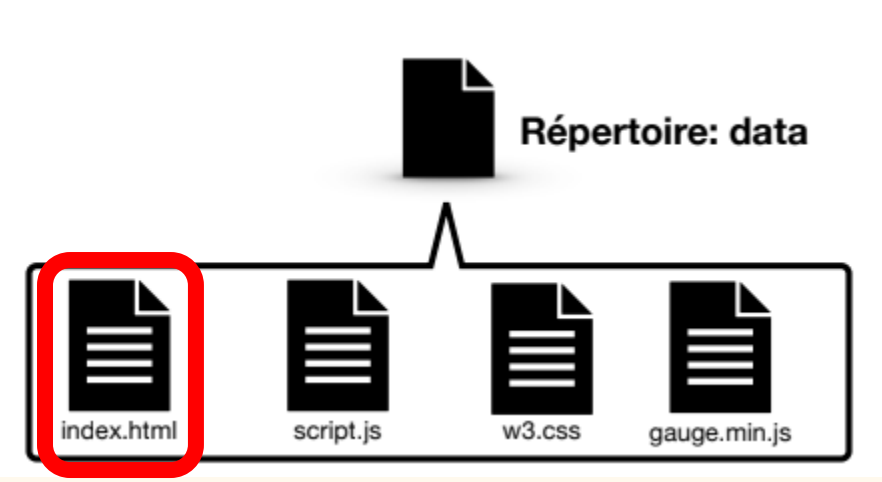
La fonction temperature :

```
float AHT20::temperature(){
    float temperature = 0.0f;
    getValues();
    temperature = _temperature;
    return temperature;
}
```

Cette fonction retourne la température mesurée par le capteur AHT20. Elle appelle d'abord la fonction getValues() afin d'effectuer une mesure à jour, puis récupère la valeur stockée dans la variable _temperature, correspondant à la température ambiante en degrés Celsius, et la renvoie sous forme de nombre flottant.

# VII- ESP32 Interface WEB

L'interface web de l'ESP32 est gérée via un serveur HTTP embarqué directement dans le microcontrôleur. Lors de l'initialisation, l'ESP32 se connecte à un réseau Wi-Fi puis lance un serveur web qui écoute les requêtes entrantes. Les pages HTML, CSS et JavaScript sont stockées dans la mémoire flash de l'ESP32 via le système LittleFS, puis servies aux clients lorsqu'ils accèdent à l'adresse IP de l'ESP32. Ce serveur peut aussi traiter des requêtes dynamiques pour afficher en temps réel des données issues des capteurs connectés (température, humidité, etc.). Grâce à cette approche, l'utilisateur peut interagir avec l'ESP32 depuis n'importe quel appareil équipé d'un navigateur, sans qu'aucune application ne soit nécessaire.




Dans cette partie nous allons traiter le fichier index.html

## VII-1. Index.html :

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="icon" href="data:;">
  <link rel="stylesheet" href="w3.css">
  <title>ESP32 WebServer</title>
  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.5.0/css/all.min.css">
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>
```

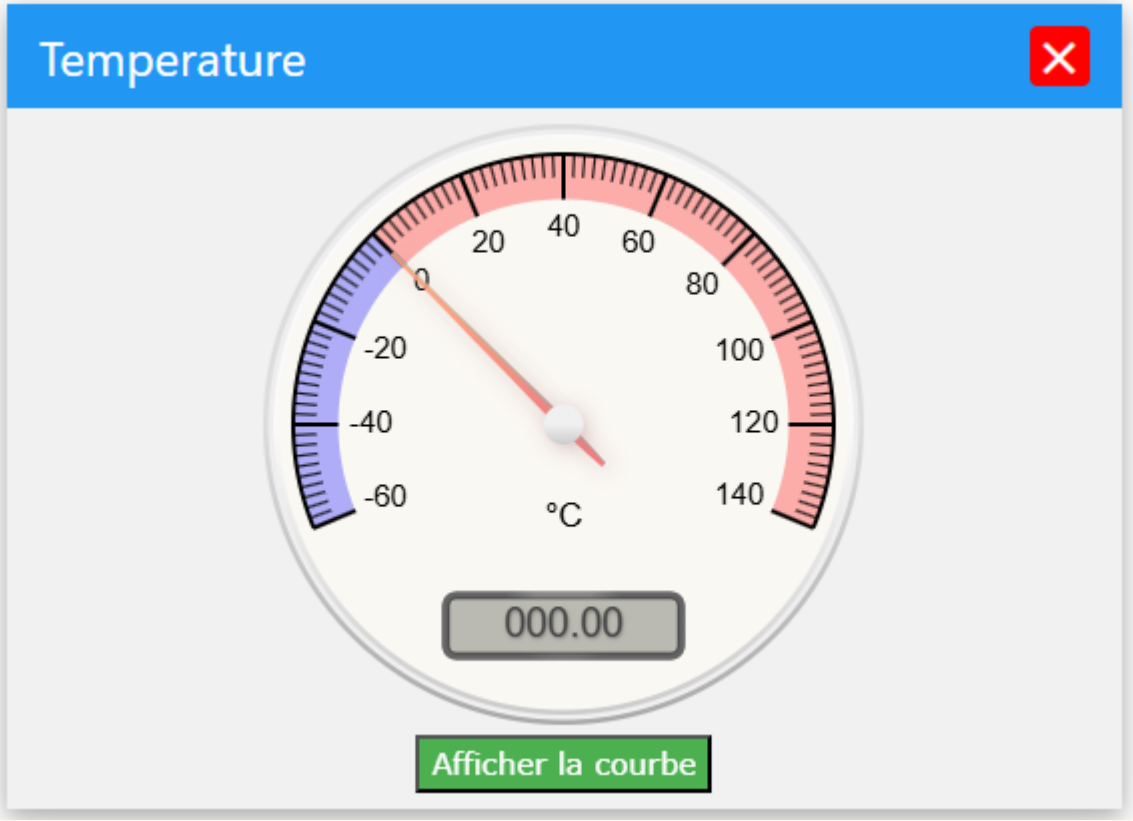
Cette section <head> du fichier HTML configure la page web servie par l'ESP32. Elle définit l'encodage des caractères en UTF-8 pour assurer une bonne gestion du texte, spécifie la compatibilité avec les navigateurs modernes, et adapte l'affichage aux différents écrans grâce à la meta viewport. Un favicon minimaliste est inclus pour éviter les requêtes inutiles. Le fichier CSS w3.css est chargé pour fournir une base de styles simple et responsive. Le titre de la page est défini pour s'afficher dans l'onglet du navigateur. Pour enrichir l'interface, la bibliothèque d'icônes Font Awesome est intégrée, permettant d'utiliser facilement des icônes vectorielles. Enfin, Chart.js est inclus pour permettre la création de graphiques dynamiques et interactifs, facilitant la visualisation en temps réel des données collectées par l'ESP32. Cette configuration garantit une interface web claire, esthétique et fonctionnelle.

Pour la mise en page, nous avons utilisé majoritairement les classes W3.CSS ainsi que du CSS classique.

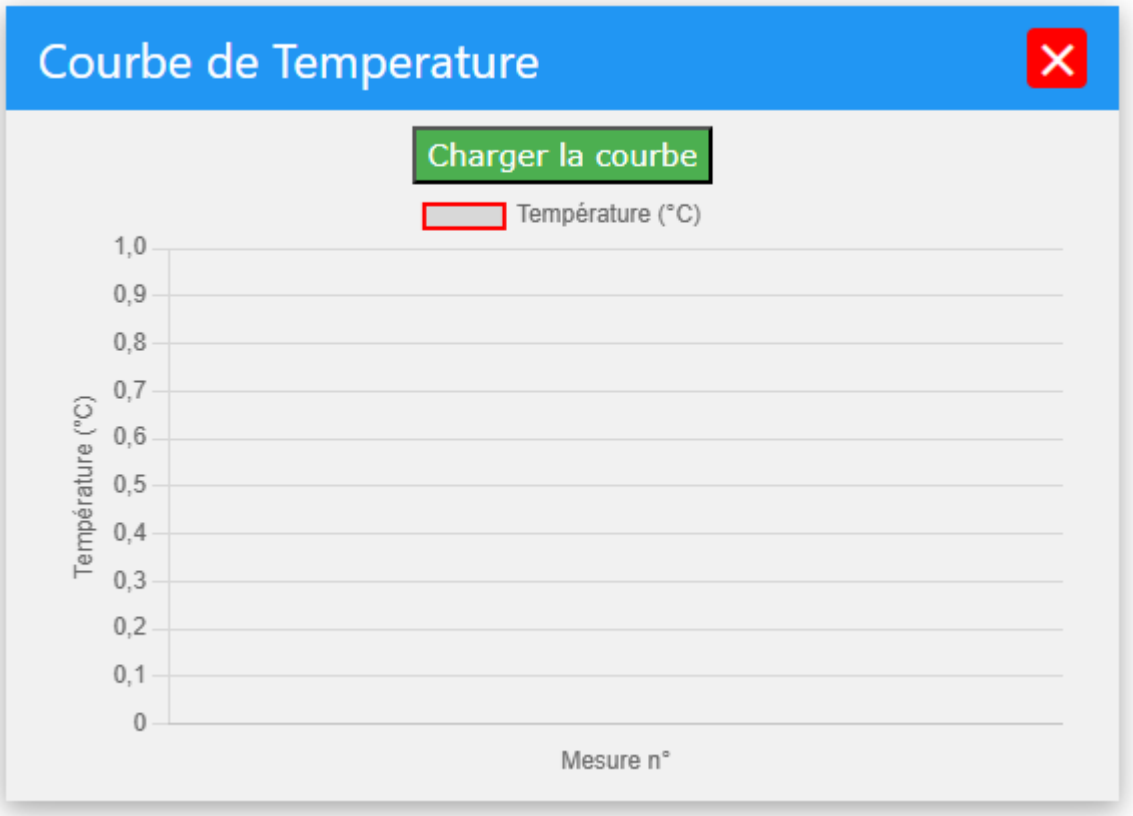
La balise <body> sera décomposée par section et par article pour une meilleure compréhension

## A. Section 1:

```
<article id="temperature" class="w3-third s4 w3-padding">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Temperature</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-light-grey w3-padding">
            <canvas id="canvas_temperature_id"></canvas>
            <button style="margin-top: 5px;" class="w3-green "
onclick="showCard('courbeTEMP')">Afficher la courbe</button>
          </div>
        </div>
      </article>
```

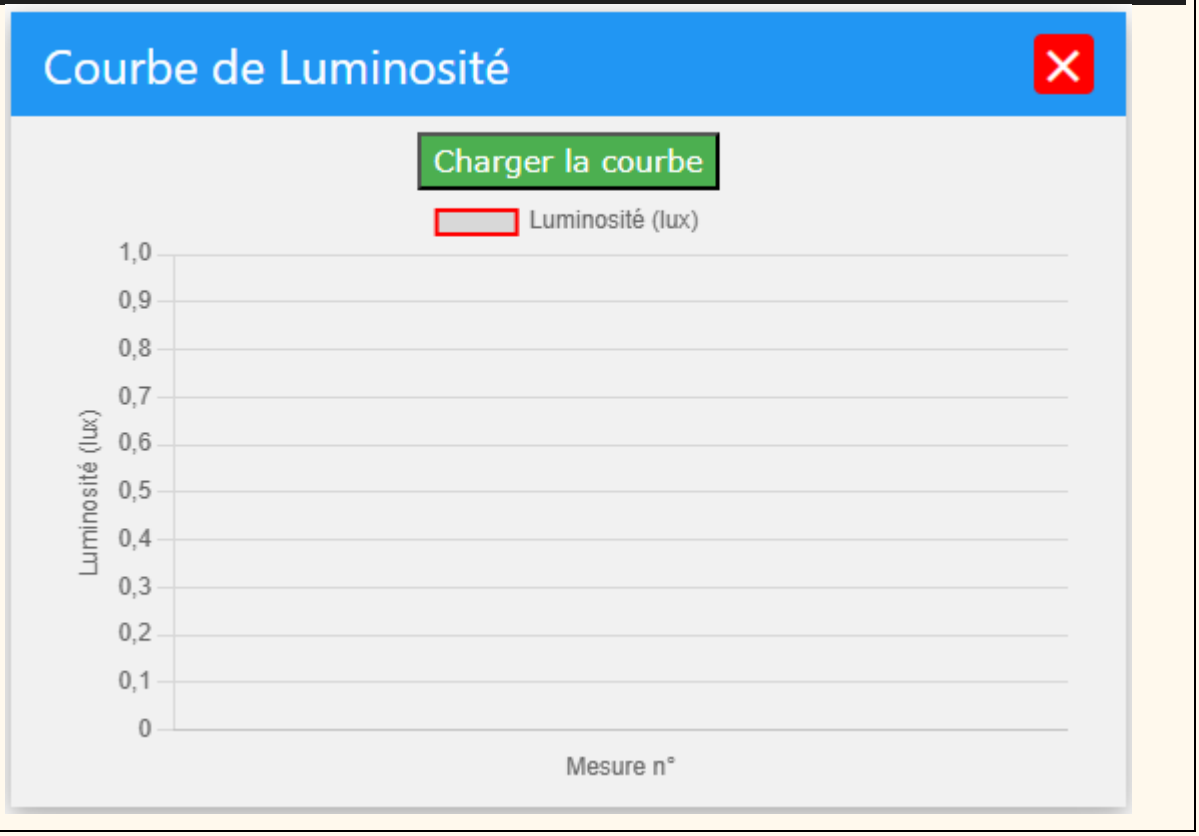


```
<article id="courbeTEMP" class="w3-third s4 w3-padding">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Courbe de Temperature</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-light-grey w3-padding">
            <button class="w3-green "
onclick="getCourbeTemperature()">Charger la courbe</button>
            <canvas id="temperatureGraph" width="500"
height="300"></canvas>
          </div>
        </div>
      </article>
```



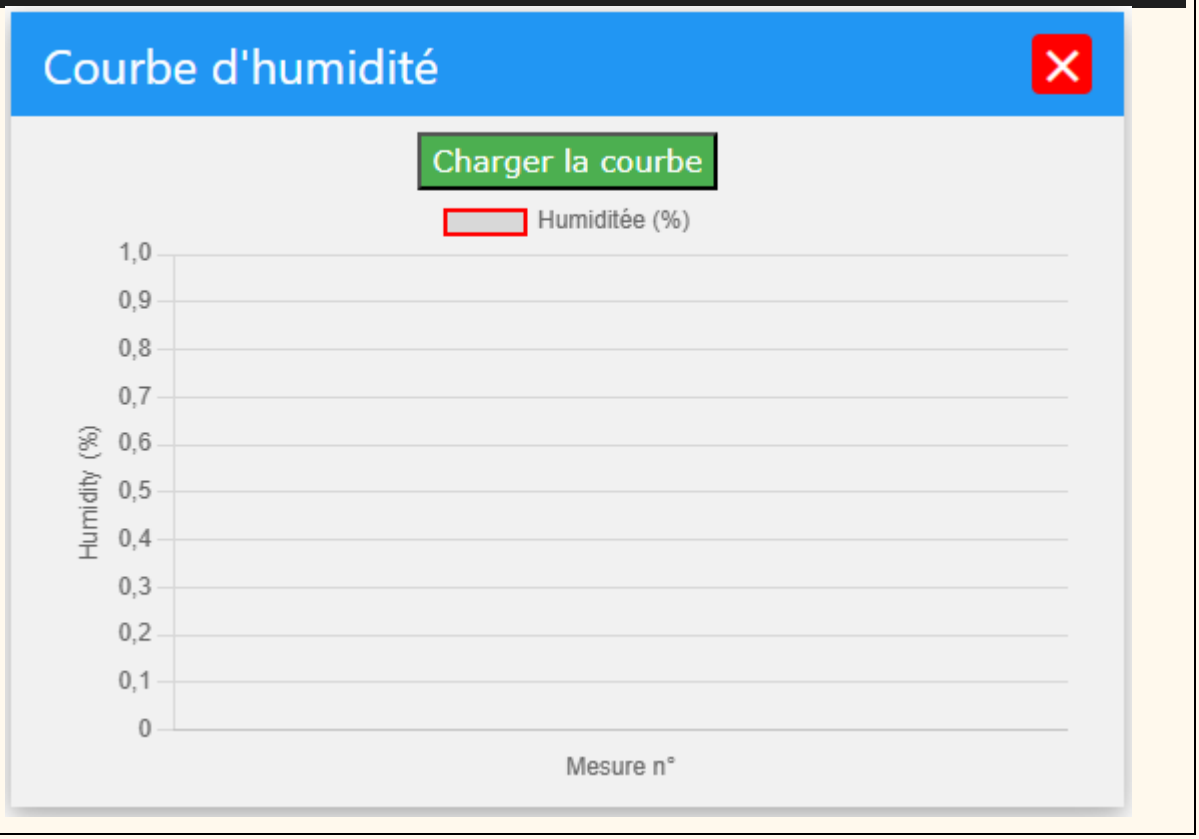
```
 <article id="illuminance" class="w3-third s4 w3-padding">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Illuminance</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-light-grey w3-padding">
            <canvas id="canvas_illuminance_id"></canvas>
            <button style="margin-top: 5px;" class="w3-green "
onclick="showCard('courbelUM')">Afficher la
                courbe</button>
          </div>
        </div>
      </article>
```

```
<article id="courbeLUM" class="w3-third s4 w3-padding">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Courbe de Luminosité</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-light-grey w3-padding">
            <button class="w3-green "
onclick="getCourbeIlluminance()">Charger la courbe</button>
            <canvas id="illuminanceGraph"
width="500"height="300"></canvas>
          </div>
        </div>
      </article>
```



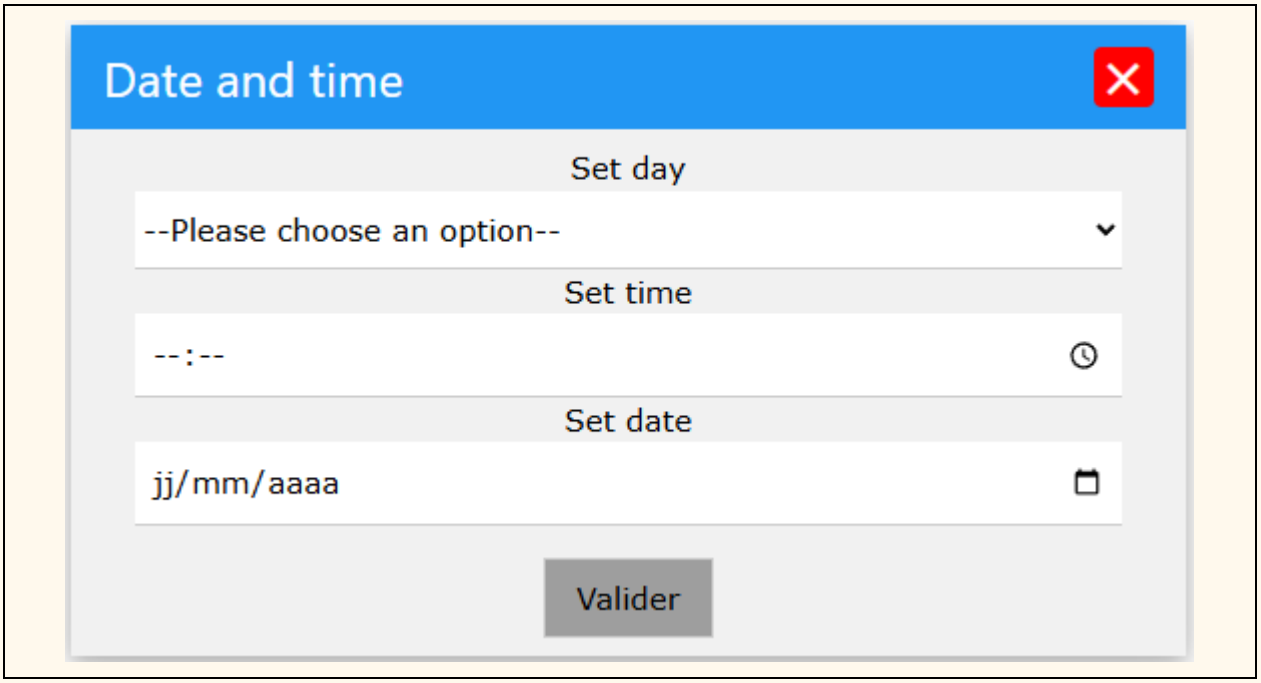
```
<article id="humidity" class="w3-third s4 w3-padding">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Humidity</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-light-grey w3-padding">
            <canvas id="canvas_humidity_id"></canvas>
            <button style="margin-top: 5px;" class="w3-green "
onclick="showCard('courbehUM')">Afficher la courbe</button>
          </div>
        </div>
      </article>
```

```
<article id="courbehUM" class="w3-third s4 w3-padding">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Courbe d'humidité</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-light-grey w3-padding">
            <button class="w3-green  "
onclick="getCourbeHumidity()">Charger la courbe</button>
            <canvas id="HumidityGraph" width="500"height="300"></canvas>
          </div>
        </div>
      </article>
```

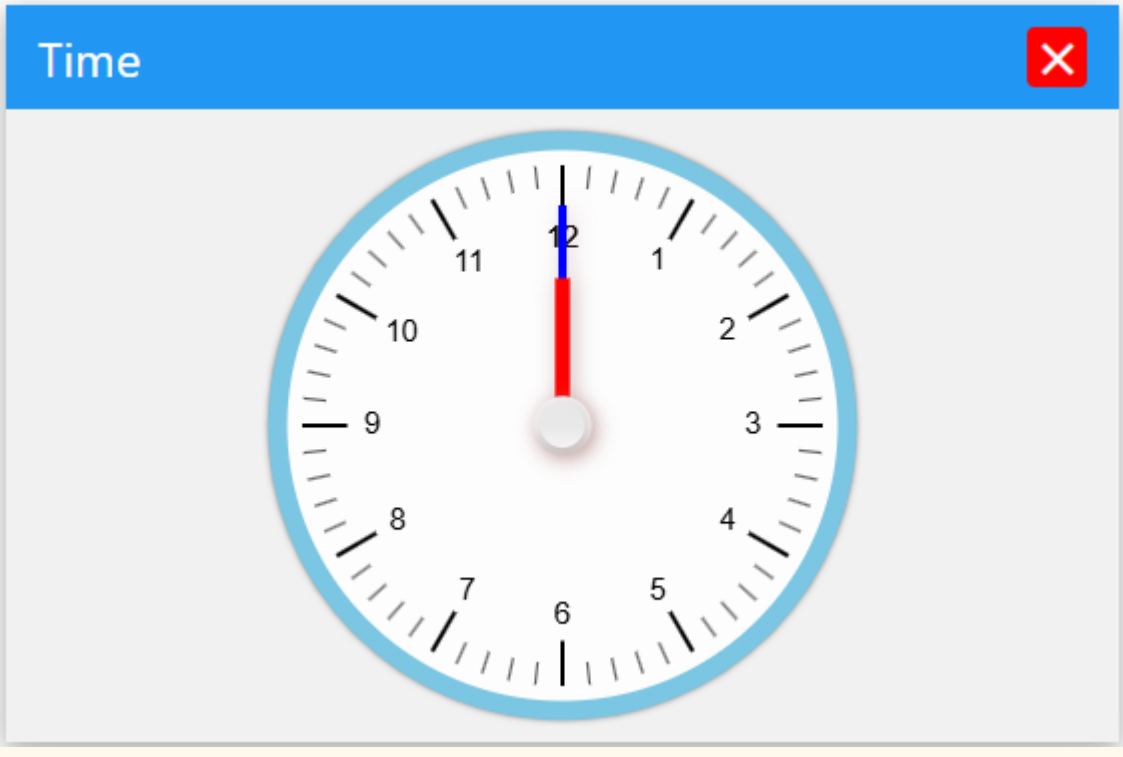


## B. Section 2 :

```
<article id="datetime" class="w3-third s4 w3-padding ">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Date and time</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-light-grey w3-padding">
            <form id="datetime-form" class="w3-container">
              <label>Set day</label>
              <select class="w3-select">
                <option value="">--Please choose an option--</option>
                <option value="0">Dimanche</option>
                <option value="1">Lundi</option>
                <option value="2">Mardi</option>
                <option value="3">Mercredi</option>
                <option value="4">Jeudi</option>
                <option value="5">Vendredi</option>
                <option value="6">Samedi</option>
              </select>
              <label>Set time</label>
              <input id="thetime" class="w3-input" type="time" required>
              <label>Set date</label>
              <input id="thedate" class="w3-input" type="date" required>
              <button type="submit" class="w3-button w3-gray w3-hover-
green w3-border w3-margin-top">Valider</button>
            </form>
          </div>
        </div>
      </article>
```

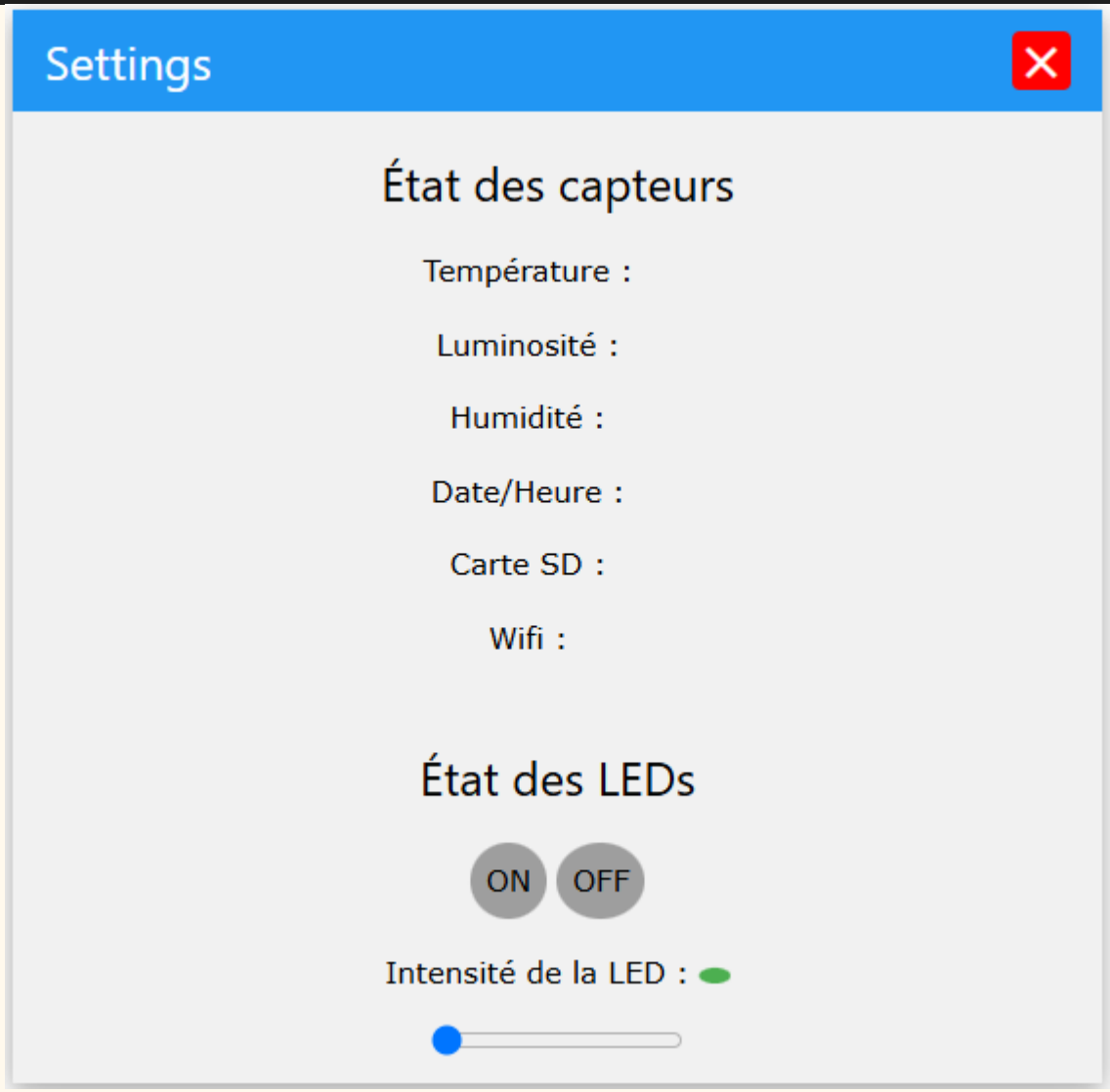


```
<article id="Time" class="w3-third s4 w3-padding ">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Time</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-light-grey w3-padding "
            style="display: flex; justify-content: center; align-items:
center;">
            <div style="position: relative; width: 300px; height:
300px;">
              <canvas id="horloge-minutes" style="position: absolute;
top: 0; left: 0; z-index: 1;"></canvas>
              <canvas id="horloge-heures" style="position: absolute;
top: 0; left: 0; z-index: 2;"></canvas>
            </div>
          </div>
        </div>
      </article>
```



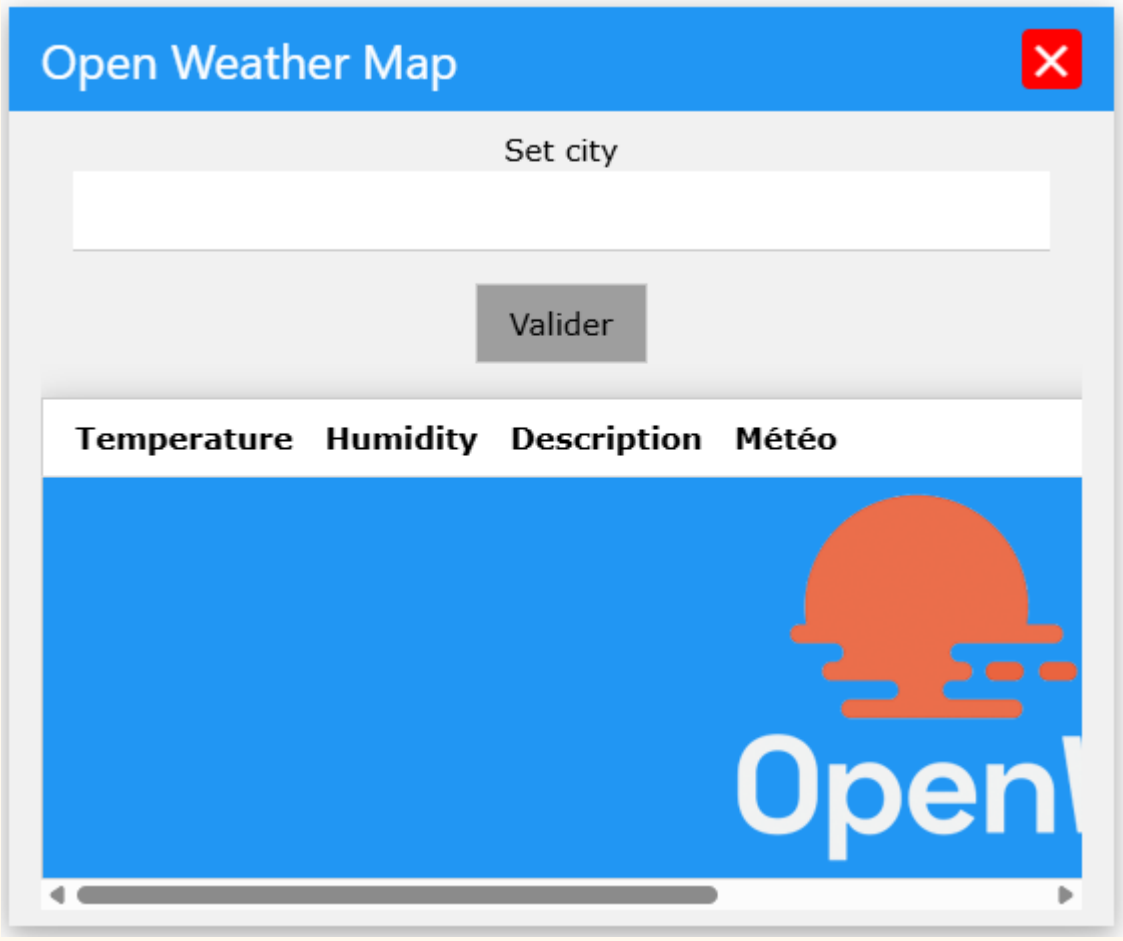
```
<article id="settings" class="w3-third s4 w3-padding ">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Settings</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-center w3-light-grey w3-padding">
            <h3>État des capteurs</h3>
            <div class="w3-section">
              <p>Température : <span id="sensor_temp"
class="voyant"></span></p>
              <p>Luminosité : <span id="sensor_light"
class="voyant"></span></p>
              <p>Humidité : <span id="sensor_humidity"
class="voyant"></span></p>
              <p>Date/Heure : <span id="sensor_datetime"
class="voyant"></span></p>
              <p>Carte SD : <span id="sensor_SDcarte"
class="voyant"></span></p>
              <p>Wifi : <span id="sensor_Wifi"
class="voyant"></span></p>
            </div>
          </div>
          <div class="w3-container w3-center w3-light-grey w3-padding">
            <h3>État des LEDs</h3>
            <div class="w3-section">
              <button id="ON_LED" class="w3-button w3-grey w3-hover-
green  w3-badge ">ON</button>
              <button id="OFF_LED" class="w3-button w3-grey w3-hover-red
w3-badge">OFF</button>
            </div>
            <p>Intensité de la LED : <span id="sensor_temp" class="w3-
badge w3-green w3-padding-small"></span></p>
            <input id="ledSlider" type="range" min="0" max="100"
value="0">
          </div>
```

```
        </div>
    </article>
```



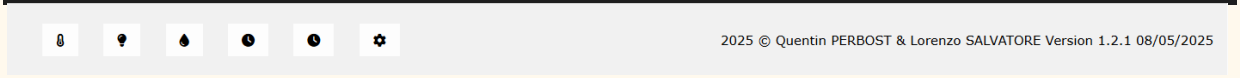
```
<article id="weather" class="w3-third s4 w3-padding ">
        <div class="w3-card-4 w3-center">
          <header class="w3-blue header-flex">
            <h1 class="header-title">Open Weather Map</h1>
            <div class="red-square">X</div>
          </header>
          <div class="w3-container w3-light-grey w3-padding">
            <form id="city-form" class="w3-container">
              <label>Set city</label>
              <input id="thecity" class="w3-input" type="text" required>
              <button type="submit" class="w3-button w3-gray w3-hover-
green w3-border w3-margin-top">Valider</button>
            </form>
            <div class="w3-responsive">
              <table class="w3-table-all w3-center w3-card-4 w3-margin-
top">
                <tr>
                  <th>Temperature</th>
                  <th>Humidity</th>
                  <th>Description</th>
                  <th>Météo</th>
                </tr>
                <tr class="w3-blue">
                  <td style="text-align: center; vertical-align:
middle;" id="temp_city"></td>
                  <td style="text-align: center; vertical-align:
middle;" id="hum_city"></td>
                  <td style="text-align: center; vertical-align:
middle;" id="lum_city"></td>
                  <td><img
src="https://openweathermap.org/themes/openweathermap/assets/img/logo_wh
ite_cropped.png"
                      id="meteo_city" /> </td>
                </tr>
              </table>
            </div>
```

```
        </div>
      </div>
    </article>
```



## C. Le footer

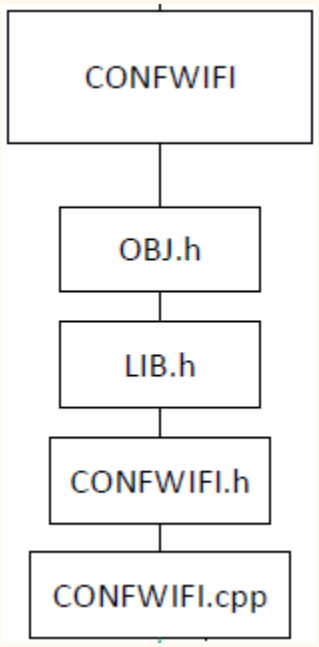
```
<footer class="navbar footer-main">
    <div>
    </div>
    <div>
      <button onclick="showCard('temperature')" class="w3-button"><i
class="fas fa-thermometer-half"></i></button>
      <button onclick="showCard('illuminance')" class="w3-button"><i
class="fas fa-lightbulb"></i></button>
      <button onclick="showCard('humidity')" class="w3-button"><i
class="fas fa-tint"></i></button>
      <button onclick="showCard('datetime')" class="w3-button"><i
class="fas fa-clock"></i></button>
      <button onclick="showCard('Time')" class="w3-button"><i class="fas
fa-clock"></i></button>
      <button onclick="showCard('settings')" class="w3-button"><i
class="fas fa-cog"></i></button>
    </div>
    <div>
      <p style="color:black"> 2025 © Quentin PERBOST & Lorenzo SALVATORE
Version 1.2.1 08/05/2025 </p>
    </div>
  </footer>
```



## VII-1. ESP32 Module WIFI

Pour la partie WIFI nous allons utiliser la bibliothèque WiFi.h

Cette partie de code se situe dans le fichier CONFWIFI.cpp :




Pour la connexion WIFI, nous avons fait 5 fonctions :

Les fonctions : function_connect, function_disconnect, function_got_ip sont les fonctions de base pour la connexion wifi avec la bibliothèque WiFi.h.

## A. CONFWIFI.h

```
#ifndef CONF_WIFI_H
#define CONF_WIFI_H

#include <LIB.h>
#include <OBJ.h>
#include <HTTPClient.h>
extern RTC3231 myRTC3231;
extern DS1621 myDS1621;
extern BH1750 myBH1750;
extern AHT20 myAHT20;
extern SSD1306 mySSD1306;
class CONFWIFI
{
public:
    // constructor
    CONFWIFI();
    void set_routes();
    bool wifi_begin(const char *ssid, const char *password);
    void initLittleFS();
    void get_value();
    void get_state(bool state_DS1621, bool state_RTC3231, bool
state_BH1750, bool state_AHT20, bool state_SDCard, bool state_WIFI);
    String httpGetRequest(String url);

private:
    const char *_ssid;
    const char *_password;
};
#endif
```

## B. CONFWIFI.cpp

### Les fonctions Wifi :

```
void function_connect(WiFiEvent_t event, WiFiEventInfo_t info){}
void function_disconnect(WiFiEvent_t event, WiFiEventInfo_t info){}
void function_got_ip(WiFiEvent_t event, WiFiEventInfo_t info){
  Serial.print("Adresse Ip : ");
  Serial.println(WiFi.localIP());
  digitalWrite(LED_BUILTIN, HIGH);
  server.begin();
  Serial.print("Serveur started on port 80");
}
```

Lorsque l’on est connecté, la led intégrée à l'ESP32 s’allume.

La fonction wifi_begin :

```
bool CONFWIFI::wifi_begin(const char *ssid, const char *password)
{
  _ssid = ssid;
  _password = password;
  WiFi.begin(_ssid, _password);
  WiFi.mode(WIFI_STA);
  WiFi.onEvent(function_connect, ARDUINO_EVENT_WIFI_STA_CONNECTED);
  WiFi.onEvent(function_disconnect,ARDUINO_EVENT_WIFI_STA_DISCONNECTED);
  WiFi.onEvent(function_got_ip, ARDUINO_EVENT_WIFI_STA_GOT_IP);
  initLittleFS();
  return true;
}
```

*voir VII-2. pour “initLittleFS();”

Cette fonction wifi_begin permet de connecter l'ESP32 à un réseau Wi-Fi en mode station (WIFI_STA). Elle commence par stocker le SSID et le mot de passe fournis, puis lance la connexion avec WiFi.begin(). Ensuite, elle définit le mode Wi-Fi sur station et attache trois événements de rappel : l’un déclenché lors de la connexion (function_connect), un autre lors d’une déconnexion (function_disconnect), et un dernier quand une adresse IP est obtenue (function_got_ip).

La fonction set routes :

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(LittleFS, "/index.html", "text/html"); });
  server.on("/w3.css", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(LittleFS, "/w3.css", "text/css"); });
  server.on("/script.js", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(LittleFS, "/script.js", "text/javascript");
});
  server.on("/gauge.min.js", HTTP_GET, [](AsyncWebServerRequest
*request)
            { request->send(LittleFS, "/gauge.min.js",
"text/javascript"); });
```

La fonction set_routes configure toutes les routes de l'interface web hébergée par l'ESP32 via le serveur AsyncWebServer. Elle établit les liens entre les URL demandées par le navigateur et les actions correspondantes dans le code. Ici, elle permet de servir des fichiers statiques comme index.html, w3.css, et des scripts JavaScript depuis le système de fichiers LittleFS.

```
server.on("/action", HTTP_GET, [](AsyncWebServerRequest *request)
            {
            if(request->hasParam("led"))
            {
              String res=request->getParam("led")->value();
              Serial.println(res);
              if(res == "1"){
                analogWrite(21, 100);}
              if(res == "0"){
                analogWrite(21, LOW);}
            }
              request->send(200); });
server.on("/set_led_intensity", HTTP_GET, [](AsyncWebServerRequest
*request)
            {
              if (request->hasParam("intensity")) {
                String res = request->getParam("intensity")->value();
                int intensity = res.toInt();
                Serial.println(res);
                analogWrite(21,intensity);
              }
              request->send(200, "text/plain", "OK"); });
```

Elle gère aussi des requêtes dynamiques : contrôle de la LED via /action et /set_led_intensity.

```
server.on("/temperature", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(200, "/text/plain",
String(temperatureValue)); });
  server.on("/humidity", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(200, "/text/plain", String(humiditeValue));
});
   server.on("/luminosite", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(200, "/text/plain",
String(luminositeValue)); });
  server.on("/courbetemp", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(200, "/text/plain",
String(temperatureValue)); });

  server.on("/courbehum", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(200, "/text/plain", String(humiditeValue));
});
  server.on("/courbelum", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(200, "/text/plain",
String(luminositeValue)); });
```

Ces routes permettent à l'ESP32 de transmettre en temps réel les valeurs des capteurs au navigateur via des requêtes HTTP de type GET. Chaque route correspond à une donnée spécifique :

- /temperature, /humidity, et /luminosite fournissent les valeurs actuelles respectivement de la température, de l'humidité et de la luminosité.
- /courbetemp, /courbehum, et /courbelum permettent de récupérer ces mêmes valeurs pour les afficher sous forme de courbes, généralement à l'aide de bibliothèques comme Chart.js.

Le client (navigateur) interroge ces routes à intervalles réguliers pour afficher des données à jour sans recharger la page, ce qui permet une visualisation fluide et dynamique des mesures environnementales.

```
server.on("/minute", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(200, "/text/plain", String(minuteValue));
});
  server.on("/heures", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(200, "/text/plain", String(heureValue)); });
  server.on("/set_datetime", HTTP_GET, [](AsyncWebServerRequest
*request)
            {
              // Vérifie si tous les paramètres sont présents
              if (request->hasParam("day") && request->hasParam("time")
&& request->hasParam("date")) {
                String day = request->getParam("day")->value();
                String time = request->getParam("time")->value();
                String date = request->getParam("date")->value();
                uint8_t hh = time.substring(0, 2).toInt();
                uint8_t mm = time.substring(3, 5).toInt();
                uint8_t index = day.toInt();
                uint16_t year = date.substring(0, 4).toInt();
                uint8_t month = date.substring(5, 7).toInt();
                uint8_t day_ = date.substring(8, 10).toInt();
                myRTC3231.setTime(hh, mm, ss);
                myRTC3231.setDate(index,day_,month,year);
                request->send(200, "text/plain", "Date et heure reçues
avec succès");
              } else {
                request->send(400, "text/plain", "Paramètres
manquants");
              } });
```

Ces routes permettent de gérer l'affichage et la mise à jour de l'heure et de la date à travers l'interface web :

- /minute et /heures envoient les valeurs actuelles des minutes et des heures stockées dans l'ESP32, souvent mises à jour à partir d'un module RTC (Real Time Clock).
- /set_datetime permet de régler l'heure et la date à distance. Lorsqu'un utilisateur soumet un formulaire via la page web, les champs day, time, et date sont récupérés. Le serveur extrait les éléments (heure, minute, jour, mois, année) puis les envoie au module RTC3231 pour mettre à jour l'horloge.

Cela permet une synchronisation manuelle du temps via le navigateur.

```
server.on("/getstatetemp", HTTP_GET, [](AsyncWebServerRequest *request)
            { request->send(200, "/text/plain", String(_StateDS1621));
});
  server.on("/getstatehumidity", HTTP_GET, [](AsyncWebServerRequest
*request)
            { request->send(200, "/text/plain", String(_StateAHT20));
});
  server.on("/getstatelight", HTTP_GET, [](AsyncWebServerRequest
*request)
            { request->send(200, "/text/plain", String(_StateBH1750));
});
  server.on("/getstatedatetime", HTTP_GET, [](AsyncWebServerRequest
*request)
            { request->send(200, "/text/plain", String(_StateRTC3231));
});
  server.on("/getstateSDcarte", HTTP_GET, [](AsyncWebServerRequest
*request)
            { request->send(200, "/text/plain", String(_StateSAVE)); });
  server.on("/getstateWifi", HTTP_GET, [](AsyncWebServerRequest
*request)
            { request->send(200, "/text/plain", String(_StateWIFI)); });
```

Chaque route retourne une information sous forme de texte indiquant si un module est actif, présent ou fonctionnel.

- /getstatetemp /getstatehumidity /getstatelight /getstatedatetime : indique si la fonction begin des capteurs respectifs à renvoyer un OK lors de la connexion (Une seule fois dans le setup).
- /getstateSDcarte : indique si la carte SD est correctement montée et prête à enregistrer des données.
- /getstateWifi : informe sur l'état de la connexion WiFi (connecté ou non).

Ces informations peuvent être affichées sous forme d'icônes colorées ou de messages dans l'interface web pour que l'utilisateur voie en un coup d'œil quels composants sont opérationnels.

L'API Open Weather Map est intégrée dans le code de la manière suivante :

- Une route est intégrée dans le but de recevoir de la part du serveur une chaîne de caractères saisie par l'utilisateur : ex : Paris, Cergy, Lisbonne.

- Cette route permet aussi d'assembler l'URL à destination de l'API en fonction de la ville, pays, régions souhaitées.

```
server.on("/set_city", HTTP_GET, [this](AsyncWebServerRequest *request)
          {
              zeCity = request->getParam("city")->value();
              Serial.println(zeCity);
              String TheURL = URL + zeCity + "&appid=" +  API_KEY +
UNITS + LANG;
              Serial.println(TheURL);
              String payload = httpGetRequest(TheURL);
              request->send(200,"application/json", payload); });
```

### La fonction httpGetRequest :

Elle réalise une requête HTTP de type GET vers une URL spécifiée en paramètre. Elle utilise un objet WiFiClient pour gérer la connexion réseau et un objet HTTPClient pour exécuter la requête. La fonction initialise la connexion HTTP avec l'URL donnée, puis envoie la requête GET. Si la réponse HTTP est positive (code supérieur à 0), le code HTTP est affiché dans le moniteur série, ainsi que le contenu de la réponse, qui est récupéré sous forme de chaîne de caractères. Enfin, la connexion HTTP est fermée proprement avant de retourner le contenu récupéré. Cette méthode permet d'interroger des services web externes (API, serveurs de données) depuis l'ESP32 et d'utiliser les données reçues dans le programme

```
String CONFWIFI::httpGetRequest(String url){{
    WiFiClient client;
    HTTPClient http;
    String payload = "";
    http.begin(client, url);
    int httpResponseCode = http.GET();
    if (httpResponseCode > 0){
      Serial.print("HTTP code:");
      Serial.println(httpResponseCode);
      payload = http.getString();
      Serial.println(payload);}
    http.end();
    return payload;
  }
}
```

## VII-2. ESP32 Module LittleFs

### A. LittleFs

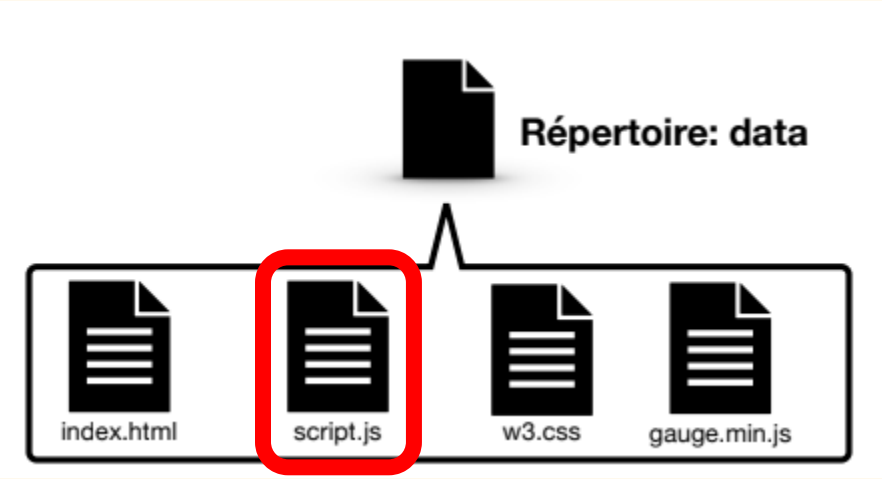
Elle se situe dans la bibliothèque CONFWIFI.

LittleFS est utilisé pour héberger les fichiers de l'interface web. Cette fonction organise proprement la configuration Wi-Fi et le système de fichiers en vue de l'exploitation web.

```
void CONFWIFI::initLittleFS()
{
  if (LittleFS.begin(true) == false)
  {
    Serial.println("Error Init LittleFs");
  }
  else
  {
    File root = LittleFS.open("/");
    File file = root.openNextFile();
    while (file)
    {
      Serial.print("File: ");
      Serial.println(file.name());
      file.close();
      file = root.openNextFile();
    }
  }
}
```

La fonction initLittleFS() initialise le système de fichiers LittleFS intégré à l'ESP32. Ce système de fichiers permet de stocker et de gérer localement des fichiers (HTML, CSS, JavaScript, etc.) qui composent l'interface web. Lors de l'appel, si l'initialisation échoue, un message d'erreur est affiché dans le moniteur série. Sinon, la fonction parcourt l'ensemble des fichiers présents à la racine du système de fichiers et affiche leurs noms un par un. Cela permet de vérifier que les fichiers nécessaires à l'interface web sont bien présents dans la mémoire flash de l'ESP32.

## VII-3. ESP32 AsyncWeb et API Fetch




Dans cette partie nous allons traiter le fichier script.js

Cette partie étant composée de fonctions similaires (par exemple : récupérer la température, la luminosité ou l'humidité), nous avons choisi de détailler uniquement le fonctionnement d'une seule fonction en expliquant ensuite les variations nécessaires pour les autres variables. Cela permet d'éviter les répétitions inutiles, tout en conservant une bonne compréhension du principe général.

En effet, le code JavaScript chargé de récupérer ces valeurs repose sur une structure commune, seule l'URL de la requête ou la variable cible change selon la donnée souhaitée (température, humidité, luminosité, etc.).

## A-Bouton de la LED

### La fonction button_on_handler :

```
async function button_on_handler() {
  try {
    const res = await fetch("/action?led=1");
    flag = true;
    if (res.ok == false) {
      throw new Error("Error response");
    }
  }
  catch (error) {
    console.log("Error:", error);
  }
};
```

Cette fonction est déclenchée lors de l'appui sur un bouton permettant d'allumer une LED connectée à l'ESP32 via l'interface web.

Elle est définie comme une fonction asynchrone (async) afin de pouvoir utiliser la méthode await, ce qui permet d'attendre la réponse du serveur avant de poursuivre l'exécution du code.

Fonctionnement :

1. La fonction envoie une requête HTTP GET vers l'URL /action?led=1, indiquant au serveur que l'on souhaite allumer la LED (le paramètre led=1 est interprété côté ESP32 comme une commande d'activation).
2. Elle utilise await fetch(...) pour attendre la réponse du serveur sans bloquer complètement le reste du script.
3. Si la réponse est correcte (res.ok == true), une variable flag est mise à true, indiquant que l'action s'est bien déroulée.

4. En cas d'erreur (réponse non valide ou problème réseau), l'exception est attrapée via un bloc try...catch et un message d'erreur est affiché dans la console du navigateur.

### La fonction button_off_handler :

```
async function button_off_handler() {
  try {
    const res = await fetch("/action?led=0");
    flag = false;
    if (res.ok == false) {
      throw new Error("Error response");
    }
  }
  catch (error) {
    console.log("Error:", error);
  }
};
```

Cette fonction est utilisée pour éteindre la LED via l'interface web.

Elle est définie comme asynchrone (async) afin de permettre l'utilisation de la commande await, qui attend la réponse du serveur avant de poursuivre.

Fonctionnement :

1. Requête au serveur : Une requête HTTP GET est envoyée à l'adresse /action?led=0. Ce paramètre (led=0) est interprété par le serveur (ESP32) comme une commande pour éteindre la LED.

2. Attente de la réponse : La fonction attend la réponse avec await fetch(...). Cela permet de gérer la réponse sans bloquer le reste du script.

3. Traitement de la réponse :
    - Si la réponse est valide (res.ok == true), une variable locale flag est mise à false, indiquant que la LED est désormais éteinte.

- Si la réponse est invalide, une erreur est levée et capturée par le bloc catch.

4. Gestion d'erreur : En cas de problème de communication ou de réponse incorrecte, un message d'erreur est affiché dans la console avec console.log("Error:", error).

## B- Animation des gauges


Les gauges ci dessus sont gérées via la fonction suivante :

### La fonction getdata :

```
async function getdata() {
  try {
    const res = await fetch("/data");
    if (res.ok == false) {
      throw new Error("Error response");
    }
    else {
      const dataValue = await res.text();
      gaugedata.value = dataValue;
      console.log(dataValue);
    }
  }
  catch (error) {
    console.log("Error:", error);
  }
};
```

Cette fonction est utilisée pour récupérer une donnée depuis le serveur via une requête HTTP et mettre à jour une jauge animée (gauge).

Elle est définie comme asynchrone (async) pour permettre l'utilisation de await.

Fonctionnement :

1. Requête GET vers /data :
   Le client envoie une requête HTTP à l'ESP32. L'URL /data est un exemple générique ici. Dans le code réel, cette URL sera remplacée par :
   - /temperature pour la jauge de température,
   - /humidity pour la jauge d'humidité,
   - /luminosite pour la jauge de luminosité.
2. Vérification de la réponse :
   - Si la réponse est invalide (res.ok == false), une erreur est levée.
   - Sinon, le texte retourné par le serveur (la valeur mesurée) est extrait avec res.text().
3. Mise à jour de la jauge :
   - La variable gaugedata.value reçoit la nouvelle valeur.
   - Cette mise à jour est automatiquement reflétée graphiquement dans l'interface utilisateur via la jauge animée.
4. Débogage :
   - La valeur reçue est affichée dans la console pour vérification.

Réutilisation :

En dupliquant cette fonction et en remplaçant simplement le chemin /data par le chemin correspondant au capteur (/temperature, /humidity, /luminosite), on obtient trois fonctions quasi identiques capables d'animer les jauges en fonction des valeurs de chaque capteur.

## C- Gestion des courbes

### La fonction getnewData :

```
async function getNewTemperature() {
  try {
    const res = await fetch("/courbedata");
    if (!res.ok) {
      throw new Error("Erreur de réponse");
    }
    const tempValue = await res.text();
    console.log("data reçue :", dataValue);
    const newdata = parseFloat(dataValue.trim());
    if (isNaN(newdata)) {
      throw new Error("data invalide reçue");
    }
    data.push(newdata);
    labels.push(data.length);
    if (data) {
      data.data.labels = labels;
      data.data.datasets[0].data = data;
      data.update();
    }
  } catch (error) {
    console.log("Erreur :", error);
  }
}
```

Cette fonction JavaScript permet de mettre à jour dynamiquement les courbes de suivi (graphique) à partir des données reçues en temps réel du serveur. Elle repose sur une requête HTTP vers l'ESP32, qui retourne les mesures actualisées du capteur.

Fonctionnement détaillé :

1. Requête vers le serveur :
    - La ligne const res = await fetch("/courbedata"); envoie une requête GET vers une URL spécifique, ici /courbedata.
    - Cette URL doit être remplacée par :
        - /courbetemp pour la température,
        - /courbelum pour la luminosité,
        - /courbehum pour l'humidité.
2. Traitement de la réponse :
    - Si la réponse n'est pas correcte (!res.ok), une erreur est levée.
    - Sinon, la valeur est récupérée avec await res.text().
3. Conversion et validation :
    - La valeur texte est nettoyée avec .trim() et convertie en nombre à virgule flottante (parseFloat).
    - En cas de conversion invalide (isNaN), une erreur est générée.
4. Mise à jour du graphique :
    - La nouvelle valeur est ajoutée au tableau data.
    - Un nouveau label est ajouté pour l'axe X (labels.push(data.length)).
    - Le graphique est ensuite mis à jour en assignant les nouvelles données à l'objet data.data.datasets[0].data et en appelant data.update().

Réutilisabilité du code :

Ce code peut être dupliqué et adapté facilement pour d'autres capteurs :

- En remplaçant courbedata dans l'URL par l'endpoint correspondant,

- Et en renommant les variables (getNewLuminosite(), getNewHumidity(), etc.).

  La fonction getcourbeData :

```
function getCourbeData() {
  if (!DataChart) {
    const ctx = document.getElementById('DataGraph').getContext('2d');
    DataChart = new Chart(ctx, {
      type: 'line',
      data: {
        labels: labels,
        datasets: [{
          label: 'Data (°C)',
          data: temperatures,
          borderColor: 'red',
          borderWidth: 2,
          fill: false,
          tension: 0.3
        }]
      },
      options: {
        scales: {
          x: { title: { display: true, text: 'Mesure n°' } },
          y: { title: { display: true, text: 'Data (°C)' } }
        }
      }
    });
  }
  if (!intervalId) {
    intervalId = setInterval(getNewData, 2000);
  }
}
```

Cette fonction initialise et met à jour dynamiquement un graphique de type courbe (Chart.js) qui affiche l'évolution des données dans le temps. Elle est utilisée pour visualiser en temps réel les valeurs mesurées par un capteur (par exemple, la température).

1. Initialisation du graphique

```
if (!DataChart) {
    const ctx = document.getElementById('DataGraph').getContext('2d');
    DataChart = new Chart(ctx, {...};
}
```

Vérifie si le graphique est déjà créé (grâce à if (!DataChart)), pour ne pas le recréer plusieurs fois.

- Utilise la bibliothèque Chart.js pour dessiner une courbe dans un canvas HTML avec l'ID DataGraph.
- Le graphique affiche :
    - Un ensemble de données (ici temperatures),
    - Avec des libellés X (labels) représentant le numéro des mesures,
    - Et des données Y correspondant aux valeurs mesurées (en °C ici).

2. Options graphiques

```
borderColor:  'red',  // ligne rouge
fill:  false,         // adoucit la courbe
tension:  0.3         // ne remplit pas sous la courbe
```

Les axes sont personnalisés avec un titre (Mesure n° pour l'axe X, Data (°C) pour l'axe Y), rendant la lecture plus claire.

3. Mise à jour régulière

```
if (!intervalId) {
    intervalId = setInterval(getNewData, 2000);
  }
```

Démarre un intervalle régulier toutes les 2 secondes.

- Appelle la fonction getNewData() (ou getNewTemperature() dans un exemple plus spécifique), qui récupère les nouvelles valeurs via le réseau et les insère dans le graphique.

Adaptabilité à d'autres capteurs

Cette fonction peut facilement être dupliquée et adaptée pour tracer l'évolution d'autres données :

- En modifiant le nom du dataset (label: 'Luminosité (lx)'),
- En changeant la couleur de la ligne,
- Et en appelant une fonction de mise à jour différente (par exemple getNewHumidity()).

## D- Gestion de l'horloge

### La fonction getMinutes :

```
async function getMinutes() {
  try {
    const res = await fetch("/minute");
    if (res.ok == false) {
      throw new Error("Error response");
    }
    else {
      const minute = await res.text();
      minuteGauge.value = minute * 6;
      console.log("minute reçu:", minute);
    }
  }
  catch (error) {
    console.log("Error:", error);
  }
};
```

La fonction getMinutes() permet de récupérer dynamiquement la valeur des minutes depuis le serveur ESP32, via une requête HTTP vers l'URL /minute.

Une fois la donnée reçue, elle est convertie en angle (chaque minute représentant 6 degrés sur un cadran circulaire, soit 360°/60). Cette valeur est ensuite transmise à une jauge graphique qui simule le mouvement de l'aiguille des minutes sur une horloge analogique.

Ce principe est utilisé pour afficher en temps réel l'heure actuelle sur une interface graphique, de manière visuelle et fluide. La jauge est ainsi synchronisée avec l'horloge temps réel embarquée (RTC).

Ce mécanisme peut être répliqué à l'identique pour afficher les heures et les secondes, en adaptant le facteur de conversion.

La fonction getHeures :

```
async function getHeures() {
  try {
    const res = await fetch("/heures");
    if (res.ok == false) {
      throw new Error("Error response");
    }
    else {
      const heure = await res.text();
      heureGauge.value = heure * 30;
      console.log("heure reçu:", heure);
    }
  }
  catch (error) {
    console.log("Error:", error);
  }
};
```

La fonction getHeures() est chargée de récupérer l'heure actuelle depuis le serveur en envoyant une requête HTTP à l'adresse /heures.

Une fois la valeur récupérée (représentant l'heure entre 0 et 12), celle-ci est multipliée par 30 afin de la convertir en degrés (puisque sur un cadran analogique, chaque heure équivaut à 30°, soit 360°/12). Cette valeur est ensuite appliquée à une jauge représentant l'aiguille des heures.

Ce fonctionnement permet de représenter visuellement l'heure sur une horloge graphique dans l'interface web. Elle est synchronisée en temps réel avec la valeur fournie par l'horloge RTC de l'ESP32.

Ce principe est similaire à celui utilisé pour les minutes et les secondes, chacun ayant son propre facteur de conversion angulaire.

## E- Gestion des fenêtre

### La fonction showCard :

```
function showCard(id) {
  const action = document.getElementById(id);
  if (action)
    action.style.display = 'block';
}
```

Cette fonction showCard(id) permet d'afficher un élément HTML spécifique identifié par son id. Concrètement, elle récupère l'élément via document.getElementById(id) et, si cet élément existe, elle modifie son style CSS pour le rendre visible en réglant sa propriété display sur 'block'. Cette fonction est utile pour afficher dynamiquement des sections ou des cartes dans une interface web selon les interactions de l'utilisateur.

La fonction document.addEventlistener :

```
document.addEventListener('DOMContentLoaded', () => {
  document.querySelectorAll('.red-square').forEach(btn => {
    btn.addEventListener('click', (input) => {
      const article = input.target.closest('article');
      if (article) {
        article.style.display = 'none';}});
  });
});
```

Ce code JavaScript attend que le contenu HTML de la page soit entièrement chargé (DOMContentLoaded). Une fois prêt, il fait ceci :

- Il sélectionne tous les éléments ayant la classe CSS .red-square.
- Pour chacun de ces éléments, il ajoute un écouteur d'événement sur le clic (click).
- Quand l'utilisateur clique sur l'un de ces boutons (btn), le code cherche l'élément parent le plus proche de type <article>.
- Si cet <article> est trouvé, il est caché en réglant sa propriété CSS display à 'none'.

cliquer sur un élément avec la classe .red-square fait disparaître (masque) la section <article> qui l'entoure. C'est un moyen simple de fermer ou cacher une carte ou un bloc dans l'interface.

## F- Gestion des formulaires

```
document.addEventListener("DOMContentLoaded", () => {
  // Attache l'écouteur au formulaire après que la page soit chargée
  const form = document.getElementById("datetime-form");
  if (form) {
    form.addEventListener("submit", submit_datetime_handler);
  }
});

document.addEventListener("DOMContentLoaded", () => {
  const form = document.getElementById("city-form");
  if (form) {
    form.addEventListener("submit", submit_city_handler);
  }
});
```

Dans ce code, on utilise deux fois l'événement DOMContentLoaded pour s'assurer que le DOM est complètement chargé avant d'attacher des gestionnaires d'événements à deux formulaires différents :

- Le premier bloc cherche un formulaire avec l'id datetime-form. S'il existe, il lui associe un gestionnaire d'envoi (submit) appelé submit_datetime_handler.
- Le second bloc fait la même chose pour un formulaire avec l'id city-form, en lui associant un gestionnaire submit_city_handler.

Ce que ça fait en pratique :

Dès que la page est prête, ces deux formulaires deviennent interactifs et exécutent leurs fonctions respectives quand l'utilisateur soumet les données (exemple : soumission de date/heure ou de la ville).

## D- gestion du slider

### La fonction slider.addEvenListener :

```
slider.addEventListener('input', async () => {
  const value = slider.value;
  console.log('Valeur du slider :', value);
  try {
    if (!flag) {
      const res = await fetch(`/set_led_intensity?intensity=${value}`);
      const data = await res.text();
      console.log('Réponse ESP32:', data);
    }
  } catch (error) {
    console.log('Erreur:', error);
  }
});
```

Ce code associe un écouteur d'événement sur un slider (curseur) qui se déclenche à chaque modification de sa valeur. Voici ce qu'il fait en détail :

- Lorsqu'on déplace le slider, la valeur actuelle est récupérée.
- Cette valeur est affichée dans la console pour suivi.
- Ensuite, si la variable flag est fausse (donc si une certaine condition est remplie), une requête HTTP est envoyée à l'ESP32 via l'URL /set_led_intensity avec la valeur du slider en paramètre intensity.
- La réponse de l'ESP32 est récupérée et affichée dans la console.
- En cas d'erreur (par exemple si la requête échoue), l'erreur est capturée et affichée dans la console.

En résumé :
Ce script permet de modifier en temps réel l'intensité d'une LED contrôlée par l'ESP32 selon la position du slider, tout en évitant d'envoyer la commande si flag est vrai (probablement pour éviter les conflits d'état).

## VIII- Ajout d'un support de stockage

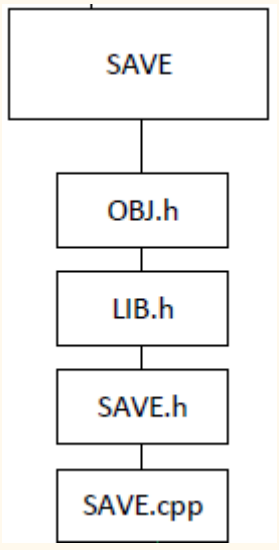
Pour gérer le stockage des informations provenant des différents capteurs, nous avons choisi d'utiliser un lecteur de carte SD interconnecté via un bus SPI. Trois bibliothèques sont utilisées pour la mise en œuvre de ce module.

Afin de faciliter son intégration dans notre programme et de permettre une évolution flexible, nous avons décidé d'encapsuler sa gestion dans une bibliothèque externe. Cette approche nous permet de modifier facilement la manière dont les données sont enregistrées, ou encore les types d'informations stockées, simplement en changeant ou adaptant la bibliothèque sans impacter le reste du code.

Les bibliothèques ci-dessous ne s'applique que pour la libraire SAVE.h

```
#include <SD.h>
#include <SPI.h>
#include <FS.h>
```

La bibliothèque SAVE.h est hiérarchiser de la manière suivante :



### A. SAVE.h

Il est organisée de la manière suivante :

```
#ifndef SAVE_H
#define SAVE_H

#include <LIB.h>
#include <OBJ.h>
#include <SD.h>
#include <SPI.h>
#include <FS.h>
extern RTC3231 myRTC3231;
extern DS1621 myDS1621;
extern BH1750 myBH1750;
extern AHT20 myAHT20;
extern SSD1306 mySSD1306;

class SAVE
{
public:
    SAVE();
    bool SD_Card_begin(uint8_t SD_SCLK,uint8_t SD_MISO,uint8_t
SD_MOSI,uint8_t SD_CS);
    void save_on_SD_card();
    void get_value_at_save();
private:
    uint8_t _SD_SCLK, _SD_MISO, _SD_MOSI, _SD_CS;
};
#endif
```

### B. SAVE.cpp

Elle est dédiée à la gestion de l'enregistrement des données issues des différents capteurs sur une carte SD. Elle permet de centraliser toutes les fonctions liées à la sauvegarde de manière modulaire et réutilisable.

L'objectif principal de cette bibliothèque est de simplifier l'écriture de données en offrant une interface claire pour :

- Créer et ouvrir des fichiers sur la carte SD,
- Écrire les valeurs des capteurs dans un format structuré (CSV, JSON, texte brut, etc.),
- Gérer les erreurs potentielles lors de l'accès au système de fichiers,
- Permettre un changement rapide de format de sauvegarde ou de structure sans modifier le programme principal.

Grâce à l'utilisation de cette bibliothèque, il est possible de modifier la façon dont les données sont archivées (fréquence, nom de fichier, format...) en ne touchant qu'à SAVE.cpp, ce qui améliore la maintenance, la lisibilité et la portabilité du code.

La fonction SD_Card_begin :

```
bool SAVE::SD_Card_begin(uint8_t SD_SCLK, uint8_t SD_MISO, uint8_t
SD_MOSI, uint8_t SD_CS)
{
    _SD_SCLK = SD_SCLK;
    _SD_MISO = SD_MISO;
    _SD_MOSI = SD_MOSI;
    _SD_CS = SD_CS;
    SPI.begin(_SD_SCLK, _SD_MISO, _SD_MOSI, _SD_CS);
    if (SD.begin(SD_CS))
    {
        Serial.println("SD Card Connectée");
        SD_Connected = true;
        uint32_t cardSize = SD.cardSize() / (1024 * 1024);
        String str = "SD Card Size: " + String(cardSize) + "MB";
        SD.remove("/datalog.csv");
        Serial.println(str);
        return true;
    }
    else
    {
        Serial.println("SD Card Déconnectée pas d'enregistrement sur la
carte SD");
        return false;
    }
}
```

La fonction SD_Card_begin est responsable de l'initialisation du module carte SD via le bus SPI. Elle prend en paramètres les broches SPI nécessaires (horloge, MISO, MOSI et CS) et configure la communication avec la carte SD.

Après l'appel à SPI.begin(), la fonction tente de monter la carte avec SD.begin(). Si l'initialisation est réussie, un message de confirmation est affiché dans le moniteur série, la taille de la carte est calculée et affichée, et un fichier datalog.csv existant est supprimé pour préparer un nouvel enregistrement. Une variable de statut (SD_Connected) est également mise à jour pour signaler que la carte est prête à être utilisée.

Si l'initialisation échoue, un message d'erreur est affiché et la fonction retourne false, indiquant que les données ne pourront pas être enregistrées sur la carte SD.

La fonction save_on_card :

```
void SAVE::save_on_SD_card(){
    if (SD_Connected){
        myFile = SD.open("/datalog_meteo.csv", FILE_APPEND);
        if (myFile){
            if (myFile.size() == 0){
 myFile.println("Température,Luminosité,Humidité,Heure:Minute:Seconde");
            }
            myFile.print(_temperatureValue);
            myFile.print(",");
            myFile.print(_luminositeValue);
            myFile.print(",");
            myFile.print(_humiditeValue);
            myFile.print(",");
            myFile.print(_heureValue);
            myFile.print(":");
            myFile.print(_minuteValue);
            myFile.print(":");
            myFile.print(_secondeValue);
            myFile.println();
            myFile.close();}
    }
}
```

La fonction save_on_SD_card() permet d'enregistrer les données des capteurs sur la carte SD, sous forme d'un fichier CSV. Elle commence par vérifier si la carte SD est bien connectée (SD_Connected). Si c'est le cas, elle ouvre (ou crée s'il n'existe pas encore) le fichier datalog_meteo.csv en mode ajout (FILE_APPEND), afin de ne pas écraser les données précédentes.

Si le fichier est vide, une ligne d'en-tête est d'abord ajoutée avec les noms des colonnes : Température, Luminosité, Humidité, et Heure. Ensuite, les valeurs mesurées courantes (_temperatureValue, _luminositeValue, _humiditeValue) ainsi que l'heure (_heureValue, _minuteValue, _secondeValue) sont écrites sous forme d'une ligne CSV, permettant un suivi chronologique des mesures.

Enfin, le fichier est refermé pour garantir la sauvegarde correcte des données. Cette méthode simple mais efficace facilite l'archivage des relevés environnementaux pour une consultation ou une analyse ultérieure sur un ordinateur.

Une évolution pertinente du système serait la mise en place d'un fichier log dédié à l'enregistrement des actions utilisateur effectuées via l'interface web. Cela permettrait par exemple de conserver une trace des commandes envoyées (activation/désactivation de la LED, réglage de l'intensité), des menus sélectionnés, ou encore des villes saisies pour la consultation météo via l'API OpenWeatherMap.

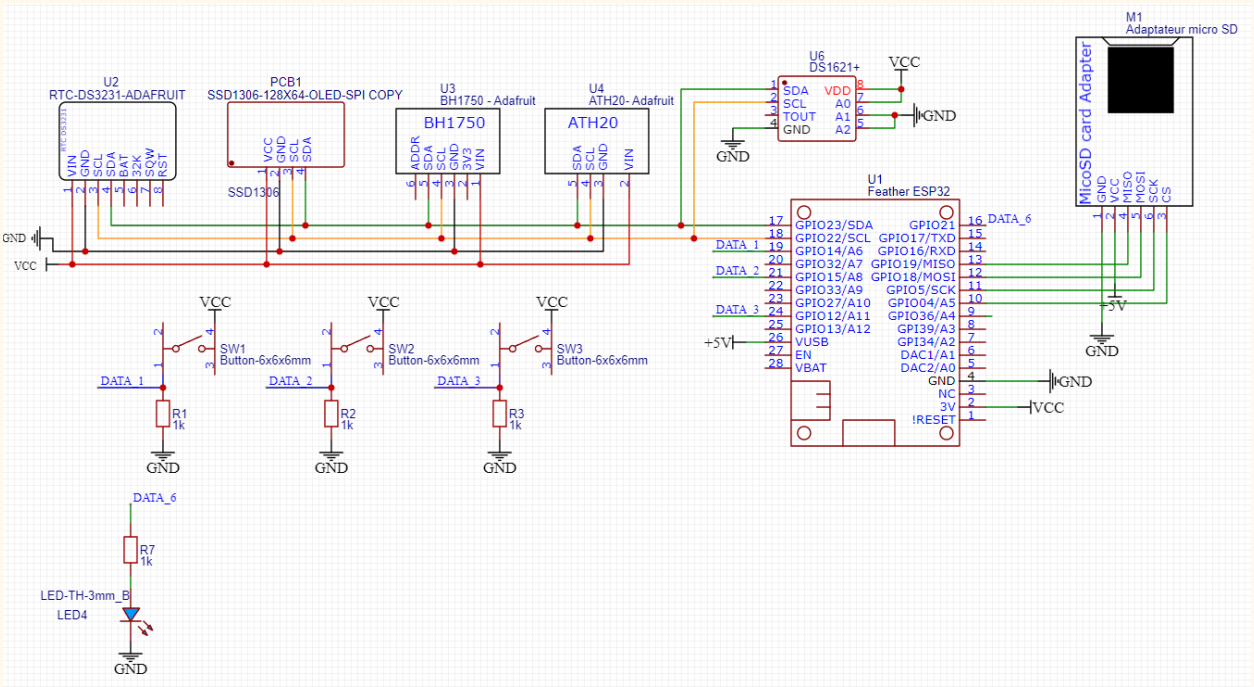
Chaque action pourrait être horodatée et enregistrée dans un fichier userlog.txt sur la carte SD. Ce fichier jouerait un rôle important en matière de traçabilité, de débogage, et éventuellement d'analyse de l'utilisation du système, notamment dans une optique d'amélioration continue ou de sécurité.

Ce journal d'activité serait facilement consultable et pourrait être traité automatiquement pour générer des statistiques ou détecter des comportements inattendus.

## IX- Conception du PCB

Pour la réalisation du pcb nous avons choisis de le faire sur l'application EASY eda,

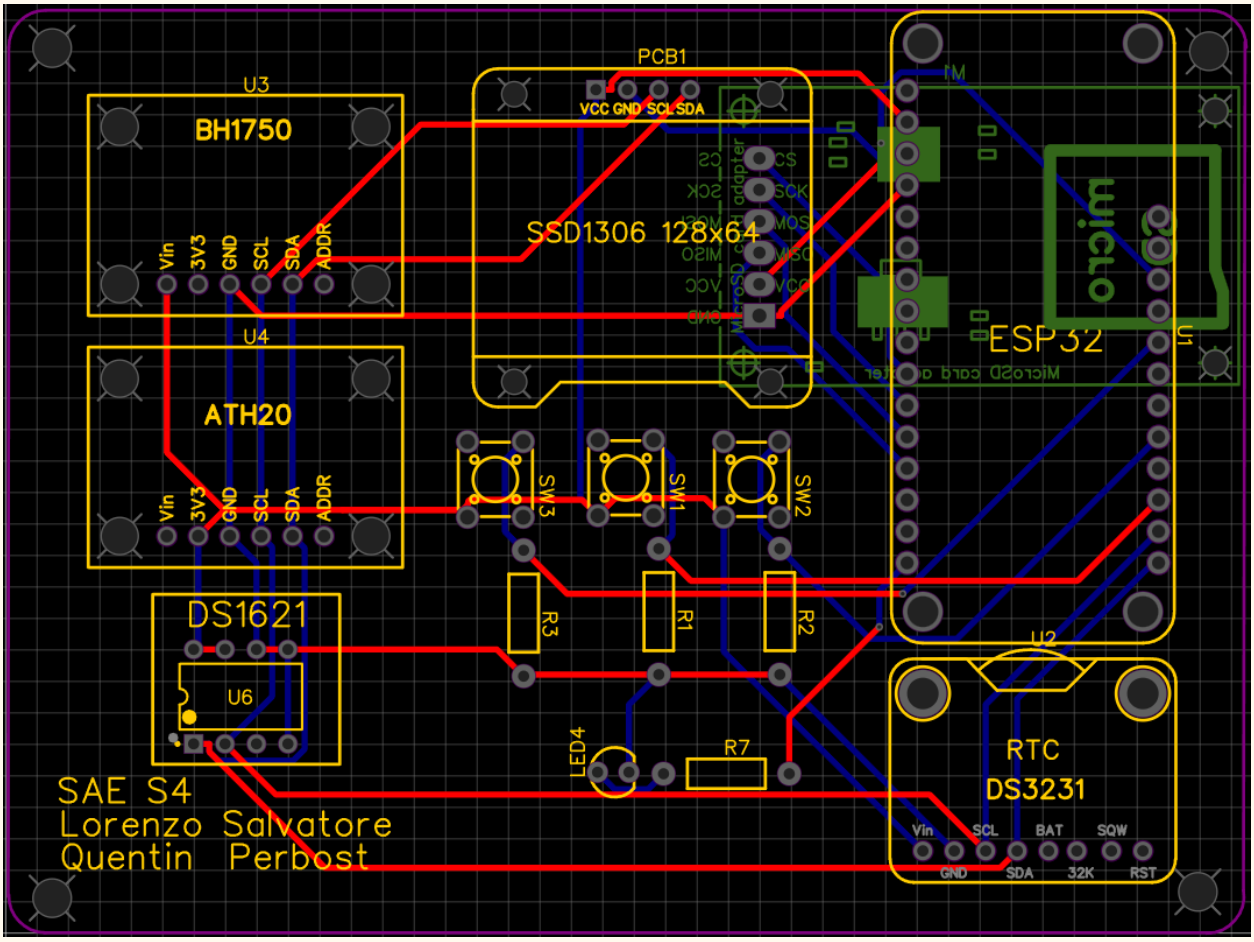
L'intégration de l'ensemble des capteurs et modules connectés à l'ESP32 a été pensée de manière claire et organisée afin de faciliter l'assemblage et assurer une bonne lisibilité du circuit. Le schéma ci-dessous présente la disposition et le câblage de tous les composants du système :




Le circuit comporte :

- Un capteur de température DS1621
- Un capteur de luminosité BH1750
- Un capteur d'humidité AHT20
- Un écran OLED SSD1306
- Un lecteur de carte SD
- 3 boutons poussoir de gestion de commande utilisateur (+ 3 résistances de PULLDOWN)
- Un ESP32 feather

La réalisation du PCB donne le routage ci-dessous :




* A noter, il y a un plan de masse sur la couche “Bottom”

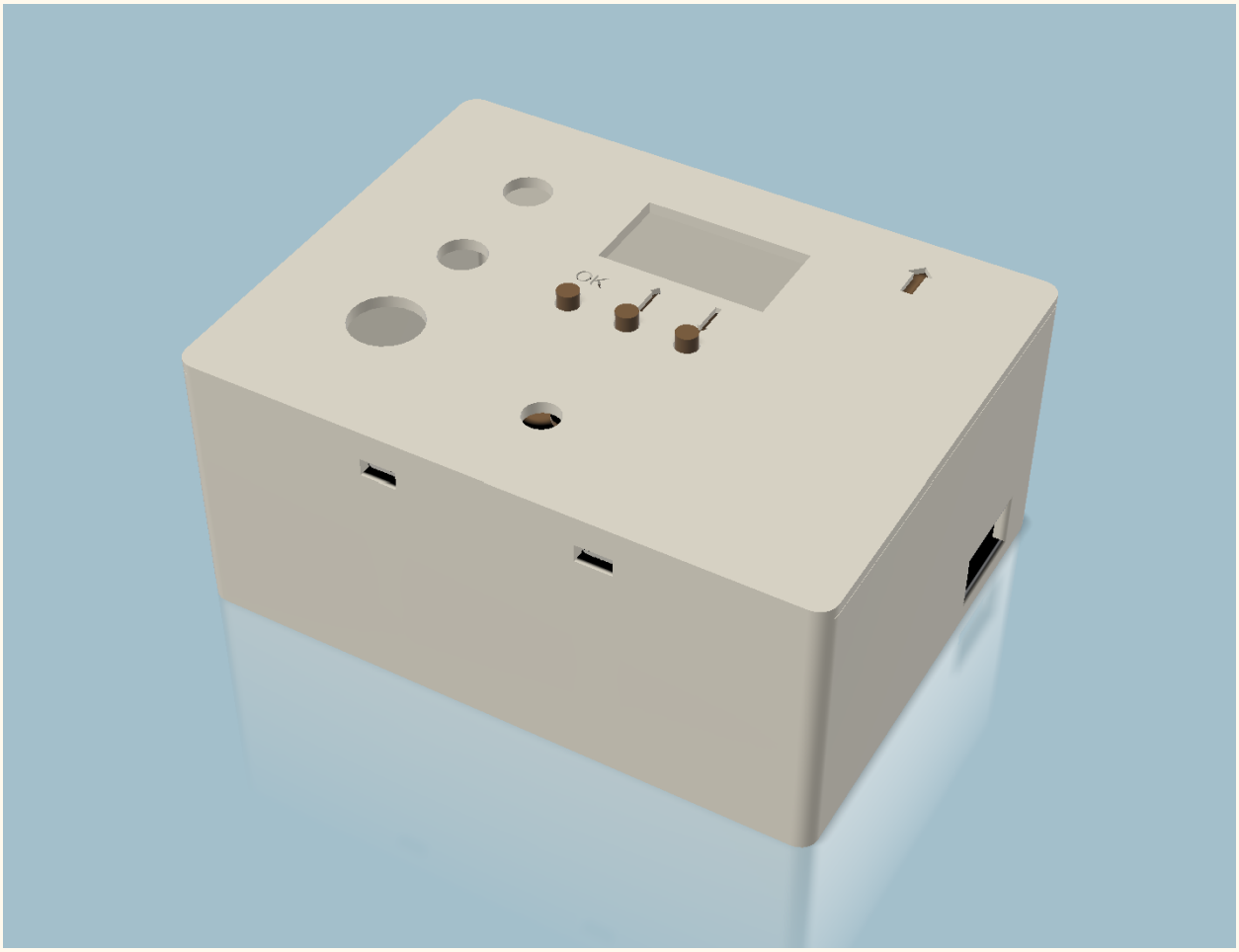
# X- Réalisation du boitier

Afin de rendre la station météo utilisable en condition réelle, c'est-à-dire dans un environnement tel qu'une maison, nous avons décidé de réaliser un boîtier.

Ce boîtier, en plus d'ajouter une valeur esthétique à la station météo, permet une utilisation simplifiée, permettant d'être utilisé par tout le monde.
De plus, il offre une protection efficace pour les composants électroniques, en évitant une manipulation directe des composants et les courts-circuits pouvant en résulter.

## A- Modélisation du boîtier

Tout d'abord, nous avons pris les cotes du PCB assemblé, puis, nous avons modélisé le boîtier sur le logiciel Fusion 360.



## B- Fabrication du boîtier

Nous avons choisi de réaliser le boîtier en plastique. Pour cela, plusieurs processus de fabrication existent, moulage, frittage sélectif par laser, stéréolithographie. Mais ces processus sont souvent coûteux et peu adaptés à la réalisation unique d'un prototype fonctionnel.

Nous avons donc choisis de réaliser le boîtier à l'aide d'un autre type de fabrication additive, la fabrication par dépôt de filament aussi appelé FDM ou tout simplement impression 3D.

Cette méthode à l'avantage d'être facile, rapide, peu chère et offre un large choix de matériaux et de couleurs.
Nous avons choisi d'imprimer le boîtier en PETG, car ce filament peu cher offre une bonne résistance mécanique, et sera suffisant pour une utilisation en intérieur (pas de nécessité de résistance aux températures faibles/élevées, UV…). De plus, le filament étant translucide, cela permet de voir la carte et les LEDs à travers le boîtier.

Le matériel utilisé est une imprimante 3D Creality Ender 3 S1 Pro, dont le firmware est modifié.